TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TANEL DETTENBORN

OPEN VIRTUAL TRUSTED EXECUTION ENVIRONMENT

Master of Science thesis

Examiner: Prof. Billy Brumley,
M.Sc. Brian McGillion
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 3st December 2014

ABSTRACT

TANEL DETTENBORN: Open Virtual Trusted Execution Environment
Tampere University of Technology
Master of Science thesis, 48 pages, 16 Appendix pages
March 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Billy Brumley, M.Sc. Brian McGillion
Keywords: Trusted Execution Environments, TEE, Security, Open Source

Hardware-based Trusted Execution Environments (TEEs) are widely deployed in mobile devices. Yet their use has been limited primarily to applications developed by the device vendors. Recent standardization of TEE interfaces by GlobalPlatform (GP) promises to partially address this problem by enabling GP-compliant trusted applications to run on TEEs from different vendors. Nevertheless ordinary developers wishing to develop trusted applications face significant challenges. Access to hardware TEE interfaces are difficult to obtain without support from vendors. Tools and software needed to develop and debug trusted applications may be expensive or non-existent.

This thesis describes Open-TEE, a virtual TEE implemented in software. Open-TEE follows GP specifications. It allows developers to develop and debug trusted applications with the same tools they use for developing software in general. Once a trusted application is fully debugged, it can be compiled for any actual hardware TEE. This thesis also describes the experience in getting trusted application developers to try Open-TEE. Open-TEE is freely available as open source[1].

[1] `https://github.com/Open-TEE/project`

TIIVISTELMÄ

TANEL DETTENBORN: Avoin Virtuaalinen Luotettu Ajoympäristö
Tampereen teknillinen yliopisto
Diplomityö, 48 sivua, 16 liitesivua
Maaliskuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Billy Brumley, DI. Brian McGillion
Avainsanat: Luotettava Ajoympäristö, Tietoturvallisuus, Avoin Lähdekoodi

Nykypäivänä lähes jokainen matkapuhelin sisältää laitteistoon pohjautuvan Luotettavan Ajoympäristön (LA, engl. Trusted Execution Envrionment). Valitettavasti ohjelmistojen kehittäminen on pääsääntöisesti rajoittunut LA laitteiston valmistajille. Asetelma on mahdollisesti muuttumassa GlobalPlatformin (GP) ponnistelun tuloksena. He ovat määritellyt ja julkaissut LA standardin, jonka tarkoituksena on standartisoida LA:n toiminnallisuus. Standartisoidin mahdollistaisi mm. kolmansien osapuolien kehitettyjen Luotettujen Ohjelmistojen (LO, engl. Trusted Application) suorittamisen eri laitteisto valmistajien LA:ssä, mutta se ei poista LO:n kehittämisen vaikeutta. Sopivan laitteiston hankkiminen ilman laitteisto valmistajan tukea voi olla hankalaa. Lisäksi LO:n kehityspakki (engl. Software Development Kit) voi olla kallis, jos se on edes olemassa.

Opinnäytetyö esittelee Open-TEE:n, joka on ohjelmallinen LA. Open-TEE on implementoitu GP standartin mukaisesti. Se tarjoaa kehittäjille hienostuneemman kehitysympäristön LA ohjelmistojen kehittämiseksi. Kun LO on valmiiksi kehitetty, se voidaan siirtää oikeaan LA:han vain kääntämällä kohde laitteistolle. Opinnäytetyö myös käsittelee tulokset, jotka on kerätty käyttäjätutkimuksella, missä kehittäjät käyttivät Open-TEE:tä LA ohjelmiston kehittämiseksi. Open-TEE on valmis käytettäväksi ja sen on julkaistu avoimena lähdekoodina Github palvelussae[2].

[2] `https://github.com/Open-TEE/project`

PREFACE

This thesis was done in collaboration with ICRI-SC Helsinki group and Intel Corporation. This work is a continuation and extension of previously published research in TrustCom 2015 (McGillion, Dettenborn, Nyman, Asokan). Following people have contributed to making this thesis:

N. Asokan (*ICRI-SC)*

Thomas Nyman (*ICRI-SC)*

Jiri Uitto (*Intel)*

Brian McGillion (*Intel)*

Billy Brumley (*Tampere University of Technology)*

I would like to express my gratitude to the people above for all the advice, comments and materials that I received in the course of writing this thesis.

Tampere, March 2016

Tanel Dettenborn

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AMD-V	AMD Virtualization
API	Application Programming Interface
CA	Client Application
CI	Continuous Integration
COW	Copy-On-Write
CPU	Central Processing Unit
DRM	Digital Rights Management
eMMC	embedded MultiMediaCard
GP	GlobalPlatform
GPL	General Public License
HMAC	keyed-Hash Message Authentication Code
HSM	Hardware Security Module
HW	Hardware
ICRI-SC	Intel Collaborative Research Institute for Secure Computing
IDE	Integrated Development Environment
Intel VT-(x,d)	Intel Virtualization
IOMMU	Input/Output Memory Management Unit
iOS	iPhone OS
IPC	Inter-Process Communication
JNI	Java Native Interface
JTAG	Joint Test Action Group
LoC	Lines of Code
NVM	Non-Volatile Memory
OS	Operating System
ObC	On-board Credentials
ODM	Original Design Manufacturer
OEM	Original Equipment Manufacturer
Open-TEE	Open Virtual Trusted Execution Environment
OS X	Mac OS X
PC	Personal Computers
PKCS	Public-Key Cryptography Standards
PSS	Proportional Set Size
RAM	Random Access Memory

REE	Rich Execution Environment
ROM	Read Only Memory
RPC	Remote Procedure Call
RPMB	Replay Protected Media Block
RSS	Resident Set Size
SDK	Software Development Kit
SGX	Intel Software Guard Extensions
SMC	Secure Monitor Call
SoC	System on Chip
SUS	System Usability Scale
TA	Trusted Application
TCPA	Trusted Computing Platform Alliance
TEE	Trusted Execution Environment
TLK	Trusted Little Kernel
TLS	Transport Layer Security
TPM	Trusted Platform Module
TUI	Trusted User Interface
USB	Universal Serial Bus
UUID	Universally Unique Identifier
VMM	Virtual Machine Manager

1. INTRODUCTION

Personal computing devices such as smartphones, tablets and laptops have become pervasive. They are used to store sensitive data and access critical services across a wide range of domains, such as banking, health care and safety, where privacy and security are paramount. On the other hand, traditional operating systems and the services that they provide are becoming so large and complex that the task of securing them is becoming increasingly harder. Hardware-based Trusted Execution Environments (TEEs) were developed to address this gap. A TEE on a device is isolated from its main operating environment by using hardware security features. It offers a smaller operating environment that provides just enough functionality so that sensitive data and operations can be offloaded to it. There is generally no need for large run-times and complex libraries with a myriad of inter-dependencies that are derived from a series of potentially 3rd party vendors, as is the case in a standard Operating Systems (OSs) such as Microsoft Windows. With the increased demand for privacy and security among users, for banking, medical and safety critical devices the need for the added security offered by the TEE is becoming paramount.

Hardware-based TEEs have been widely deployed in mobile devices for over a decade [14]. TI M-Shield [5] and ARM TrustZone [4, 3] are early examples, followed by newer architectures like the Intel SEP security co-processor [22] and Apple's "Secure Enclave" co-processor [2]. Business requirements such as the need to enforce Digital Rights Management (DRM) and subsidy locks, as well as regulatory requirements like cloning and theft protection have been the driving forces behind such large scale deployment [14]. Such requirements continue to appear: e.g. fingerprint scanners with hardware protection, hardware-backed keystores, and the recent "kill switch" [26] bill in California mandating that a mobile device must be capable of being rendered inoperable if it is stolen.

Although the early Hardware Security Modules (HSMs) like the IBM cryptocards[1]

[1] `http://www.ibm.com/security/cryptocards/`

were programmable [11], the vast majority of HSMs used with personal computers and servers today are typically application-specific modules or fixed function co-processors like the Trusted Platform Modules (TPMs) [39]. In contrast, TEEs in mobile devices are programmable. However, despite widespread deployment of hardware-based TEEs in mobile devices, application developers have lacked the interfaces to use TEE functionality to protect their applications and services. Nor have they been researched extensively in the academic community. Recent efforts by GlobalPlatform (GP) [17] to specify standard interfaces for TEE functionality in mobile devices [16] will partially address this problem. However, there are a number of factors that stand in the way of widespread use of hardware-based TEEs in application development and research. Chief among them is the difficulty of developing applications for TEEs. Software development kits for TEE application development are often proprietary or expensive. Debugging low-level TEE applications either requires expensive hardware debugging tools, or leaves the developer with only primitive debugging techniques like "print tracing" (e.g., using printf statements in C to keep track of how values of variables change during program execution).

This thesis argues that a virtual, standards-compliant TEE implemented entirely in software will allow developers to build TEE applications using tools and development environments that they are *already* familiar with. It will also allow applications to be tested and refined even when developers do not have access to devices where hardware TEE functionality has been made accessible to them. Such a facility will greatly ease TEE application development and can trigger new ways of using TEEs. We make the following contributions:

- The design and implementation of such a **virtual TEE, called Open-TEE,** which conforms to GlobalPlatform Specifications. Identifying requirements that would make Open-TEE acceptable to developers and make specific design choices informed by these requirements (Section 3). Open-TEE is publicly available on GitHub.[2]

- Show that Open-TEE is **efficient, hardware-independent** and allows a developer to carry out much of the development life cycle of standard-compliant TEE applications using popular application development environments they are already familiar with. It demonstrate that Open-TEE **significantly improves the ease-of-use** of TEE application development by conducting a

[2] `http://open-tee.github.io/`

small-scale, yet rigorous, user study with experienced professional TEE developers (Section 4).

Given the demonstrable usability benefits, this thesis recommends that organizations that develop applications for TEEs should consider incorporating Open-TEE into their development process. This thesis also hopes that this work will enable more researchers to discover the power of TEEs and use Open-TEE to develop and experiment with new TEE applications.

Note that Open-TEE is *not intended to emulate* a particular hardware TEE. The goal of Open-TEE is that a TA (Trusted Application) developed successfully with Open-TEE is guaranteed to compile and run on any target GlobalPlatform-compliant TEE.

In order to support this demanding market chipset vendors have long supported extensible TEEs in mobile devices; where implementations such as Intel's SEP security co-processor [22] and ARM's TrustZone security mode [3] are common examples. Traditionally, conventional operating environments relied on services offered by TPMs and HSMs, however, extensible TEEs are starting to be employed in these environments also.

Some sectors have been pushing forward the use cases for the TEE, most notably the media industry's requirement for DRM protected content, operators protecting their money stream by protecting the simlock and device unlock functionality. Recently, however, there have been a few requirements such as the hardware backed keystore, fingerprint scanners that are starting to push the boundaries of the services that have been traditionally offered.

The common criteria shared by all TEEs are the ability to perform some computation in a environment independent of the main operating system. To ensure this, there is generally the need for some form of hardware separation, e.g. a separate machine, a dedicated co-processor or a special mode of operation in the core that restricts access to resources depending on the state of the core. Traditionally TEEs have come in varying formats generally designed to defend against a specific threat or fill a specific niche e.g. application-specific co-processors are examples of such commonly deployed environments.

However, these systems can be very inflexible, being designed to perform only a

particular task. In some areas such as mobile devices there has long existed the need for more flexibility. Mobiles have a number of factors influencing their design and usage. Chipset vendors, Original Equipment Manufacturers (OEMs), operators (carriers) and end users each having their own set of requirements. These may be defined by regional variation in regulations which mandate certain functionality in order to achieve certification in that region. An example of such a regulation is the recent "kill switch" bill [34] in California which mandates that a mobile device must be capable of being rendered inoperable if it is stolen. Although this feature had gained popular acceptance from a number of large OEMs before the bill was even proposed, it now mandates that all manufactures wishing to sell devices in California must offer this feature.

2. BACKGROUND

The concept of using device hardware for supporting operation system security is by no means a new invention [25]. As early as in the 1970's the Cambridge CAP computer [32] was developed, which provided hardware support for this kind of feature. In the 1990's the first standardization efforts took place for hardware-assisted security, when Trusted Computing Platform Alliance (TCPA) defined a hardware-based security element for Personal Computers (PCs). Almost simultaneously, the first large scale deployment of hardware-assisted security took place in the mobile field, when mobile device manufacturers added hardware-based integrity protection of booted system software and the means for isolated execution of trusted programs. With this technology it was possible to implement, for example, subsidy locks. Extensible hardware-assisted security is not widely deployed in PCs due to the nature of the PC, which started as an open system and thus did not have the same revenue streams that warrant the inclusion of a subsidy lock. In addition the end users did not expect the same level of predictability and reliability, e.g. there has been no need for the secure storage of the calibration data as is required for safe efficient running of a mobile phone.

Today's mobile phones provide us with many different kinds of services. For example, we can browse the web and run third-party applications; actions which can be achieved even on today's feature phones. For this reason, mobile operating systems have grown in size and therefore are increasingly susceptible to software vulnerabilities [14]. However, end users still expect a certain level of predictability and reliability so there is a need for additional security. This is the reason why there is a need to use hardware assisted TEE functionality. Today almost every smartphone and tablet contains a TEE, such as ARM's TrustZone [4].

Mobile devices equipped with TEEs have the potential to replace, amongst others, wireless tokens for opening doors in buildings or cars, traditional credit and debit cards for transactions and even report on a patient's health to a medical center.

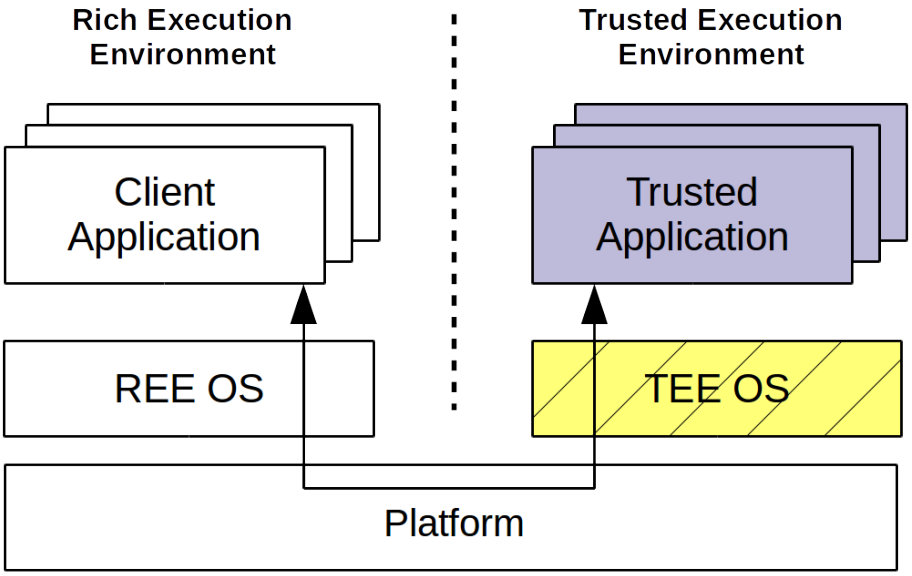

Figure 2.1 *A TEE in a Computing Device*

Recent standardization efforts in GlobalPlatform could soon make it possible for TEE functionality to be accessed in a standardized manner. GlobalPlatform has announced a number of TEE specifications [17]. The introduction of these standards have the potential to make it easier for different users to learn, practice and study this domain in a safe and controlled manner.

2.1 Trusted Execution Environment

A TEE is a secure, integrity-protected processing environment, consisting of processing, memory and storage capabilities. Figure 2.1 shows how a device can be visualized as a series of distinct environments with their own set of features and services. What follows, using the terminology introduced by GlobalPlatform [18], describes the concepts illustrated in Figure 2.1[1].

Rich Execution Environment (REE)

The word "rich" here refers to an operating environment that is feature rich, as one would expect from modern platforms such as Android, iOS, Windows, Linux or OS

[1]Figures, in this thesis, use a consistent color coding: Yellow hatched represents a TEE environment, blue a TA and black with white text is GlobalPlatform API 2.4.1. The purpose is to highlight how components are mapped to each other.

X. In some literature this environment may be referred to as the "Normal World" in reference to the fact that it is where the majority of applications are being developed for and deployed to.

Trusted Execution Environment (TEE)

The TEE is a combination of features, both software and hardware, that isolate the execution of tasks from the REE. These environments have a limited set of features and services as they are intended to only address the security critical subset of an application's functionality such as offloading some cryptographic operations or key management.

Trusted Application (TA)

An application encapsulating the security-critical functionality to be run within the TEE. This may be a service style application that provides a general feature, such as a generic cryptographic keystore, or it could be designed to offload a very specific part of an application that is running in the REE, such as a portion of the client state machine in a security protocol like TLS.

Client Application (CA)

CAs are ordinary applications (e.g. browser or e-mail client) running in the REE. CAs are responsible for providing the majority of an application's functionality but can invoke TAs to offload sensitive operations.

Examining a typical TEE application workflow sequence, using GP terminology, let's consider a common use case for TEEs: the offloading of DRM protected content. The CA would be responsible for the majority of the tasks associated with viewing the content i.e. opening the media file, providing a region in the display into which it can be rendered (the window) and providing a mechanism by which to start, stop and rewind the media. A TA would be used to decrypt the protected media stream and make the decrypted content available directly to the graphics hardware that is responsible for rendering and displaying the stream.

2.2 Use of TEE

Utilizing a TEE is not a silver bullet for securing a device. It provides defense in depth and helps narrow down the attack vectors that an attacker can leverage to compromise a device or user. Properly offloading tasks to TEE can efficiently protect sensitive data from leaking, however, if we assume the following scenario: a user has downloaded a new update for their device which contains malicious code. If that malicious application can masquerade as a legitimate CA, then the attacker could have free use of the sensitive data stored in the TEE. That is they would be able to e.g. decrypt data or sign messages as a legitimate user, however, the TEE would still ensure that the key was not revealed. Even with this limitation the benefits of using a TEE far outweigh the risks of not using it and a TEE is critical to the proper functioning of certain use cases that have become commonly available in mobile devices, such as:

Keystore: Used for storing cryptographic tokens, keys or certificates, into a TEE to make it more difficult to extract them from the device. When tokens are deployed to a TEE, a CA can make use of them through a keystore Application Programming Interface (API) and these tokens are thus not exposed to user space or Random Access Memory (RAM).

Secure storage: This can serve as a multi-purpose facility which could allow an application to store everyday information such as user identity, pictures or documents. Most implementations provide both confidentiality and integrity protections. In fact modern storage media such as embedded MultiMediaCard (eMMC) may contain a special partition, called the Replay Protected Media Block (RPMB), to assist with integrity protection. It relies on a keyed-Hash Message Authentication Code (HMAC) for its operation and this key is protected by the TEE. Mobile phone vendors have used secure storage for calibration data and firmware configurations for many years to protect their assets and the safety of end users. For example an attacker, in this case the legitimate owner of a device, may wish to alter the modem configuration to allow them to have a greater share of the bandwidth or a higher priority on the network. This can lead to poor service for other users or be potentially harmful to the user as the new settings may pose a health risk. In order to mitigate this risk the manufacturer would store these critical configurations in

secure storage and would potentially disallow the device to boot if the calibration data has been tampered with.

Secure boot is an extension to what has just been discussed, it provides the ability to measure the integrity of certain code and data during the boot and can be designed to disallow boot if any of the components have been altered. It does this by storing a list of known good configurations, i.e. the signatures of firmware and software components and comparing these to the components as they are prepared for loading. Modern implementations of secure boot extend from the hardware all the way to the user space. Non-Volatile Memory (NVM) such as the Read Only Memory (ROM) code or key hashes stored in physical write once fuses can be used to provide the basis of security. The systems can be thus designed that each verified component can be relied upon to provide the verification of the layer(s) above it. A simplified example would be that the ROM code verifies the bootloader, which in turn verifies the main OS, which in turn verifies the overall REE before launching the first user space application e.g. init. Secure boot does not guarantee that the device is free of security issues; rather it can certify that the components that have booted are the best known configuration as provided by a trusted source, e.g. the OEM.

Digital Rights Managements (DRM): One of the driving forces behind the wider adoption of TEEs has been the media industry. DRM content is becoming ubiquitous, common examples are music files from iTunes[2] or a video stream from HBO[3]. In order to protect the media stream from piracy the data is encrypted with a key that is generally device or session specific. The TEE is used to protect this key, perform the decryption of the session and make the data available to other parts of the system so it can be securely displayed.

Although the previous use cases are more prevalent on mobile devices they are now seeing widespread deployment on a variety of devices ranging from desktops to smart watches. This thesis hopes to outline that although these are some of the most common uses of a TEE, it is by no means an exhaustive list and in fact TEE technology is underutilized [14].

[2] `https://apple.com/itunes/`
[3] `https://hbo.com`

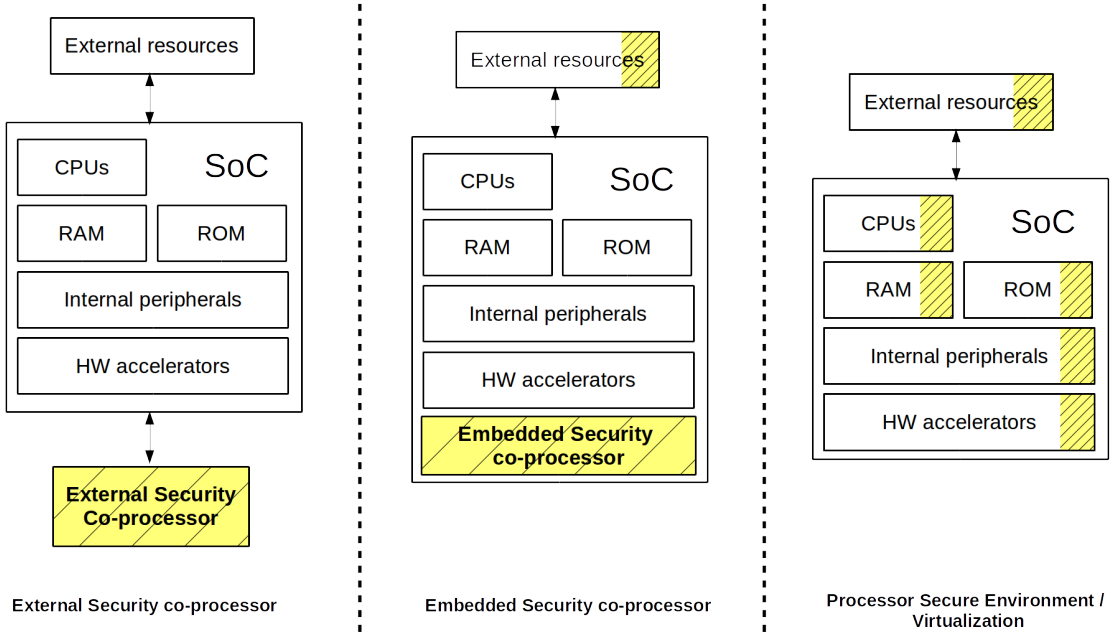

Figure 2.2 *Three potential architectural options for realizing a TEE architecture (adapted from [18])*

2.3 TEE Architectures

A TEE can be realized in different ways, but the overall concept stays the same. Figure 2.2 shows a number of ways in which these TEEs can be realized:

2.3.1 Co-Processor

A separate core, generally with its own peripherals, is used to offload the security critical tasks from the main operating environment. The benefits of such a configuration are that the operation can generally be completely isolated and it can run simultaneously with the main core. The drawback is that there is an overhead associated with transferring the data to and from the core. Also, the co-processor is generally less powerful than the main core. The co-processor design can be further separated into two alternatives:

External security co-processor is a discrete hardware module outside the physical chip (commonly referred to as "System on Chip" or SoC) containing the main core, and is thus completely isolated from it, not sharing any resources with it.

Embedded security co-processor is embedded into the main SoC and thus has the capability to share some of the resources of the main system. It is still isolated from the main processor.

2.3.2 Processor Secure Environment

Many popular mobile TEE architectures follow a configuration where a single core supports multiple virtual cores that are mutually exclusive of one another i.e. when one is running the other is suspended. Generally there is some form of trigger to allow the core to switch from one state to the other. This configuration is sometimes referred to as the "processor secure environment" [12].

ARM TrustZone is an example of this configuration. In TrustZone, the processor core can be in one of two "worlds": a "secure world" (for the TEE) and a "normal world" (for the REE). A special instruction called Secure Monitor Call (SMC) can be executed to trigger the processor running in normal world to enter "monitor mode" that marshals the transition to secure world [4]. The advantage of this configuration is that there is no need to offload the data to and from the secure world. However, there is a cost associated with having to store and restore the device state on entry and exit from a given mode. On single core devices there is also an added security benefit from having only one world running at a given time in that it ensures that the normal world OS cannot interfere with the secure world directly or indirectly (e.g., software side-channel attacks). However, this also has the disadvantage that when one world is active the other world must be completely halted, thus complicating interrupt handling and potentially causing a transition back to handle the interrupt before the task is complete.

Intel Software Guard Extensions (SGX) [29, 21] is another example of such a variant, the core does not perform a full transition to and from a secure world. Instead parts of a standard application, both code and data, are protected by mechanisms in the core. Parts of the application, called an Enclave, are encrypted by a key that is only accessible to the Central Processing Unit (CPU). When an "enter enclave (EENTER)" [23] instruction is received the code and data are decrypted and operated upon in the core. They never leave the CPU package unencrypted, thus protecting them against external access. The benefits are that there is no need

to transfer data back and forth between cores or to setup complicated transitions to and from a secure world, and there is no additional need for a separate operating environment as is required in other styles of TEE configuration. Most security use cases related to normal world applications running can be supported in this fashion by allowing small select parts of the application to be secured, thus enabling developers to move away from the paradigm of "Splitting Trust" [6] towards a model that developers are more familiar with, where the application can be self-contained. Obviously this does not remove the need for the developer to know which parts of an application need to be secured and thus which parts to run in an enclave, however, it does remove the need to develop a separate application that will run within a different environment; which in the case of a co-processor may have a completely different set of tools and requirements for the developer to learn in order to utilize the facility.

2.3.3 Virtualization

Virtualization based on hardware features such as AMD Virtualization (AMD-V) and Intel Virtualization (Intel VT-[x,d]) have existed for many years and are used extensively to provide separation of resources between different operating environments especially in high density server configurations. They rely on processor support to allow virtualization of instructions and access to resources e.g. through the use of an IOMMU (Input/Output Memory Management Unit) access to and from peripheral devices can be restricted. Though in and of themselves they are not designed solely to provide a TEE, there is recent research [9, 28] to see how these can be used as an alternative to dedicated hardware based TEEs. When deployed as TEE environments they generally rely on a Virtual Machine Manager (VMM) to provide the marshaling of access to the resources. There has been extensive research and security auditing of VMM technology in recent years [4], to the point where it is now uncommon to hear of exploits against the VMM itself. Vulnerabilities such as venom [10] highlight that there is still always the possibility of vulnerabilities in any sufficiently large code base, however, in contrast to traditional hardware based TEEs there is generally an open disclosure of the vulnerabilities and a software patch is, in many cases, a sufficient remedy.

[4]The Invisible Things lab have been active in this field. http://blog.invisiblethings.org/papers/

2.4 Standardizing TEE Functionality

The landscape for TEEs has been very diverse, with a variety of different architectural options from multiple manufacturers.

Even platforms using the same type of TEE are often not interoperable. For example, an application written for one TrustZone-based platform will generally not run on a different TrustZone-based platform. They may be using different TEE OSs or different REE OS drivers. On the other hand, developers and others who are higher up in the software ecosystem are less concerned with intricacies of low-level software or hardware but more concerned with their ability to use the capabilities of TEEs easily and across different platforms. This calls for standardization.

2.4.1 GlobalPlatform

One initiative in TEE standardization has been undertaken by GlobalPlatform [17], which "is a cross industry, non-profit association which identifies, develops and publishes specifications that promote the secure and inter-operable deployment and management of multiple applications on secure chip technology" [15]. GP offers specifications in three areas: smartcards, back-end support systems and devices. This thesis is concerned with specifications from the device working group related to the APIs for TAs.

Figure 2.3 shows the primary interfaces standardized by GP. The GP TEE Core API provides an extensive set of features such as a crypto API and secure storage that can be used to implement a TA, for example a DRM decoder. The GP TEE Client API is a very generic and thin layer consisting of a small number of functions and definitions that allow the transfer of data back and forth from the REE to a TA. A CA, for example a DRM player, will implement all complex but non-critical functionality by itself, but use the GP TEE Client API to invoke the corresponding TA, such as the DRM decoder. Between the "GP TEE Client API" running on the REE and "GP TEE core API" running on the TEE we have an effective Remote Procedure Call (RPC) mechanism where a process running in the REE can invoke tasks in the TEE.

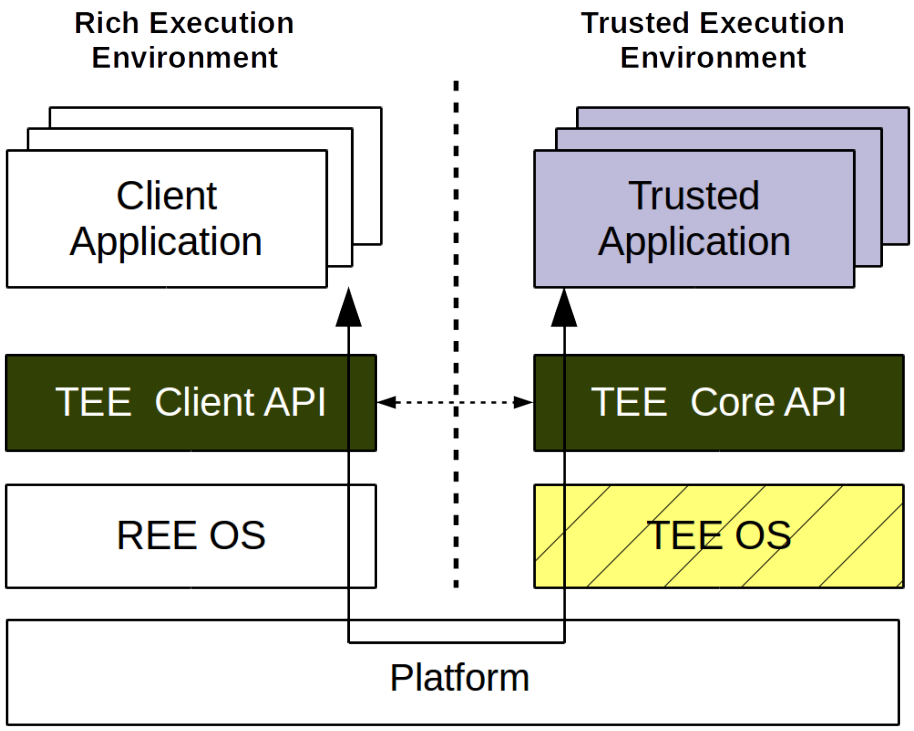

Figure 2.3 *The main APIs specified by GlobalPlatform [18]*

2.4.2 Benefits of TEE Standardization

These standardization efforts in GlobalPlatform could resolve the issue of interoperable TEEs. In other words, TEE application developers could re-use the same application across different TEEs rather than developing for a specific TEE. For example, a keystore service is already provided by Intel SEP and ARM TrustZone, but unfortunately both define their own APIs, which forces the implementation of tailored solutions on each platform.

If the TEE vendors were to agree on requirements and standards, in addition to committing to deploy and use them in their products it may provide an incentive for the developer community to utilize these features. A wider research and developer community could be a valuable resource as they will help to drive the next revolution in TEE use cases. It is already abundantly clear that there is a consumer need for more security, privacy and identity protection being among their top concerns so we can expect more demand for TEE functionality to come. Even taking existing standards and enabling them, such as implementing the Public-Key Cryptography Standard #11 (PKCS#11), in a readily accessible way can have a large immediate

impact.

PKCS#11 defines a generic interface to cryptographic tokens e.g. how to sign data. It is a well-known and used standard though it generally relies on 3rd party hardware, such as smartcards, smartcard readers, Universal Serial Bus (USB) dongles etc. Numerous languages provide wrappers for the API, many more applications are PKCS#11 aware [5] and can be configured to offload cryptographic operations to a PKCS#11 implementation. Providing this, integrated, as part of an existing device will free the end user from having to know which hardware to carry around with them and will enable application developers to tailor their applications knowing that this feature is readily available.

However, given all the benefits that standardization brings it does not remove the obstacle of gaining access to the requisite hardware nor does it simplify the task of developing and testing TAs. The remainder of this thesis focuses on how to overcome this lack of access and enable developers and researchers to gain valuable experience with TEE technology and especially its concepts.

[5] http://en.wikipedia.org/wiki/List_of_applications_using_PKCS_11

3. OPEN-TEE

Hardware-assisted TEEs have been available in mobile devices for almost a decade, but access to this technology has been granted only to privileged developers. For example, developers working for chip vendors, OEMs and Original Design Manufacturers (ODMs). Limited access for third-party developers can be accounted for by a variety of reasons, the technology is proprietary, lack of trust of the third party, lack of an easily deployed Software Development Kit (SDK). In addition, there is no unified means to access the resources and functionality of the TEEs.

In order to pave the way for the widespread use of TEE functionality by developers and researchers this thesis defines an architecture and SDK. It is implemented as a framework atop a set of tools that are familiar to the developer, thus removing the need for specialized hardware and the overheads that it incurs.

3.1 Motivation

Chapter 1 alluded the difficulties in developing TEE applications. This section will expand the four points which most influenced this thesis.

Enable developer access to TEE functionality

For a variety of reasons, access to TEEs is generally restricted to developers working for chip manufacturers and for the OEMs that make devices based on these chips. Usually, the technology is proprietary and easily deployable SDKs are not available. Furthermore, TEEs may not have a security architecture within them to safely allow complete outsiders access to the them without impairing overall security. However, there have been attempts to address this problem [24].

Provide a fast and efficient prototyping environment

The most common methods of debugging TAs are to either use expensive Joint Test Action Group (JTAG[1]) debugging or resort to primitive "print tracing" by inserting diagnostic output in the source code. The former generally allows for detailed instruction level debugging. However, the costs associated with these debuggers can be prohibitively expensive, and the setup complex. Print tracing as a debugging technique is cumbersome and clutters up the source code even to locate the source of a problem. Another concern encountered by TEE developers is that if a TA running on actual device hardware crashes, a hard reset of the device maybe required to recover, thereby significantly increasing the time and effort of debugging.

Promote research into TEE services

Ways to isolate TEEs from REEs are reasonably well understood as we saw in Section 2. What is less well understood are the types of services that could benefit from using TEEs. As the app store model[2] has proven, given an opportunity, the developer community at large is capable of pushing the boundaries and exploring new and novel ways to use technology. Making it possible for researchers to easily develop TAs could trigger the development of novel and innovative applications.

Promote community involvement

The prerequisite for involving the developer community and researchers at large is to allow them access to a freely and easily available development environment, SDK and a platform with which to experiment. The financial and technical aspects of making hardware TEEs available for development on a large scale motivates the need for a software framework for TA development which is not bound to any particular hardware or vendor.

To get the community involved, application developers need development environments, SDKs and a platform with which to experiment. Although the GP standard simplifies conceptualizing a TEE and the functionality that can be offered by it, there

[1] Joint Test Action Group standard addresses debugging of integrated circuits

[2] In this context the app store is digital distribution platform for mobile applications. An example of this could be Google Play.

are a number of obstacles that the developer must first overcome. The hardware is complex and expensive to design and manufacture, by its nature it is complicated to deploy, test with and there are no standard tools with which to work.

Safely exposing TEE functionality to application developers will enable them to innovate these novel approaches to improve the security and privacy of their applications. Exposing TEE technology to a wider audience in no way guarantees that all security threats will disappear, however, a community can provide more varied research and ideas than a small group ever could. This is one of the few ways that we can start to realize the potential of a TEE. The financial and technical aspects of accessing a hardware TEE make this infeasible so we propose the creation of a TEE framework, that is not bound to any hardware or any particular vendor, yet tries to conform to one standardization effort, for which we have chosen GP.

3.2 Requirements

Motivated by the above discussion, our aim is to develop an SDK and framework that allows for the development and testing of standard-compliant TEE applications. The framework should allow development of GP-compliant CA and TA functionality without having to rely on any particular hardware support. Open-TEE is intended to be a fast prototyping and development environment that also provides a platform from which to conduct further research into TEE functionality. Our fundamental design principle is that it should require as little configuration and maintenance as possible, allowing the developer to focus on the task at hand.

We identify the following criteria by which we can measure our TEE framework's usefulness and hence its potential success in addressing the issues that motivated it.

- **Compliance:** Our framework should comply with GP's main interfaces, the GP TEE Client and GP TEE Core APIs.
- **Hardware-independence:** As a software based solution our framework should not be dependent on a particular TEE hardware environment. It should also not be dependent on any particular hardware for the development system itself.
- **Reasonable performance:** To be readily deployed, our framework must not suffer from code bloat that adds to the on-disk footprint nor to the memory

consumption required to run it. In addition the start-up and restart times of the environment, especially that of the CAs and TAs should not be excessive. One of the perceived benefits of our framework is its ability to support fast prototyping and as such, any time penalty that is incurred will diminish its usability and the satisfaction of using it.

- **Ease-of-use:** The solution should be easily deployed and configured. It should use tools that are widely available making it more attractive (e.g., there should be no need for extra package/tool configuration on the development system).

Previous points are going to be walked through in Chapter 4 and evaluated based on the success of the Open-TEE project.

3.3 Architecture

The following section describes our design and implementation of such a software framework which we call Open-TEE. It will begin with an overview of the structure of the Open-TEE environment. Figure 3.1 identifies the main components and their relationships. The color code used in Figure 3.1 is the same as that used for Figure 2.3 to make the correspondence between the Open-TEE implementation architecture and the GP conceptual architecture is clear. Each component is described in detail below.

Base

Open-TEE is designed to function as a daemon process in user space. It starts executing Base, a process that encapsulates the TEE functionality as a whole. Base is responsible for loading the configuration and preparing the common parts of the system. Once initialized Base will fork and create two independent but related processes. One process becomes Manager and the other, Launcher which serves as a prototype for TAs.

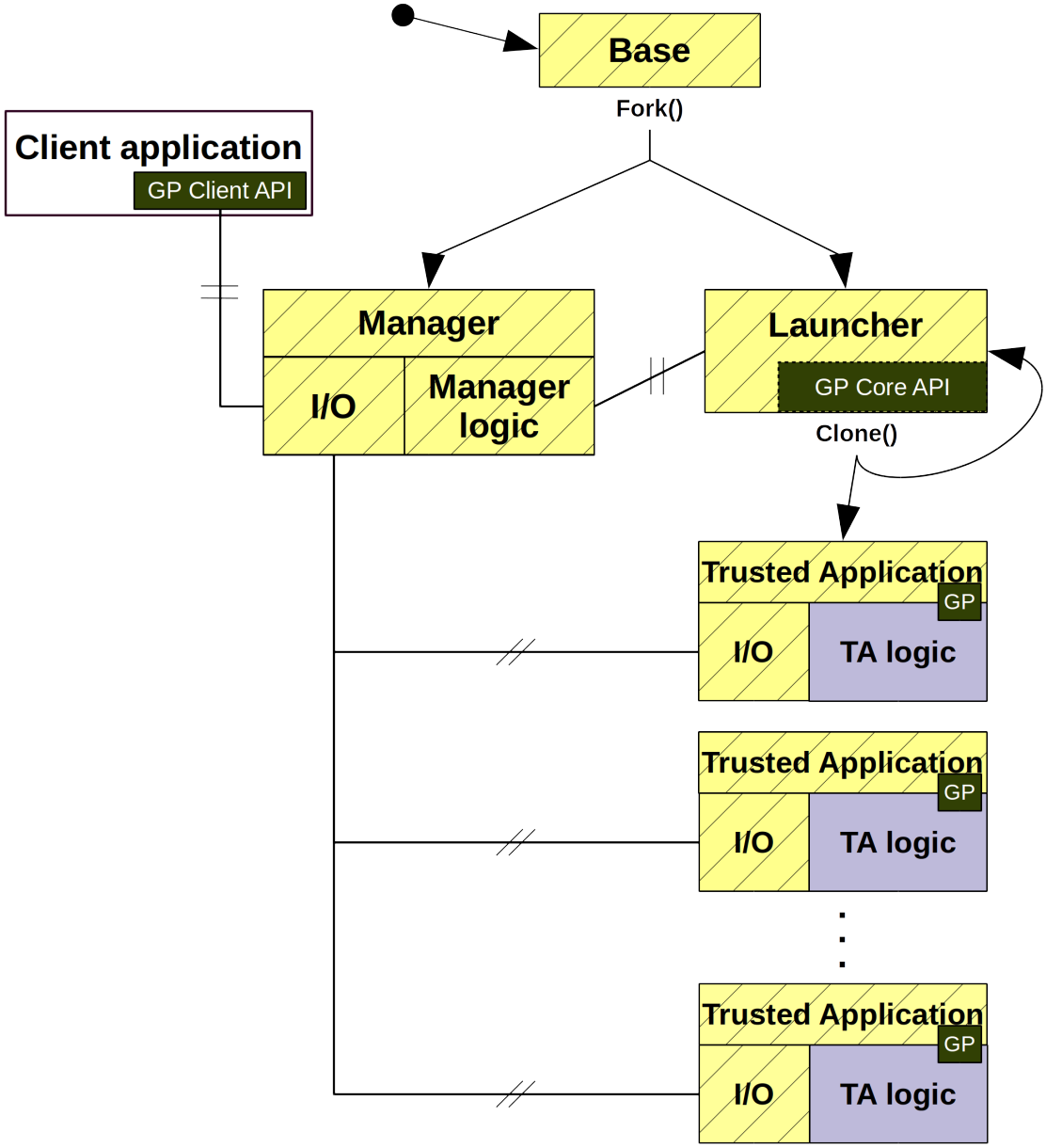

Figure 3.1 *Open-TEE architecture*

Manager

Manager can be visualized as Open-TEE's "operating system". Its main responsibilities are: managing connections between applications, monitoring TA state, providing secure storage for a TA and controlling shared memory regions for the connected applications. Centralizing this functionality into a control process can also be seen as a wrapper abstracting the running environment (e.g. GNU/Linux)

and reconciling it with the requirements imposed by the GP TEE standards. GP requirements and the host environment's functionality are not always aligned. For example, GP requirements stipulate that if a TA/CA process crashes unexpectedly, all shared resources of the connected processes must be released. In a typical running environment, this requires additional steps beyond just terminating the process. For example all shared memory must be unregistered – this needs to be a distinct action from normal process termination.

Launcher

The sole purpose of Launcher is to create new TA processes efficiently. When it is first created, Launcher will load a shared library implementing the GP TEE Core API and will wait for further commands from Manager. Manager will signal Launcher when there is a need to launch a new TA (for example, when there is a request from a CA). Upon receiving the signal, Launcher will clone itself. The clone will then load the shared library corresponding to the requested TA. The design of Launcher follows the "zygote" design pattern (such as that used in Android [1]) of preloading common components. This is intended to improve the perceived performance of starting a new TA in Open-TEE: because shared libraries and configurations common to all TAs are pre-loaded into Launcher, the time required to start and configure the new process is minimal. A newly created TA process is then re-parented onto Manager so that it is possible for it to control the TA (so that, for example, it can enforce the type of GP requirements discussed in the paragraph above).

TA Processes

The architecture of the TA processes is inspired by the multi-process architecture utilized in the Chromium Project [38]. Each process has been divided into two threads[3]. The first handles Inter-Process Communication (IPC) and the second is the working thread, referred to respectively as the I/O and TA Logic threads. This architectural model enables the process to be interrupted without halting it, as occurs when changing status flags and adding new tasks to the task queue. Additional

[3]The architecture of Manager follows the same division

benefits of this model are that it allows greater separation and abstraction of the TA functionality from the Open-TEE framework.

GP TEE APIs

The GP TEE Client API and GP TEE Core API are implemented as shared libraries in order to reduce code and memory consumption. In addition loading the GP TEE Core API into Launcher when it is created will help to reduce the startup times of the TA process removing the need to load it for every TA; this is one of the key benefits of the "zygote" design.

IPC

Open-TEE implements a communication protocol on top of Unix domain sockets and inter-process signals as the means to both control the system and transfer the messages between the CA and TA.

3.4 Implementation and Tooling

This section highlights interesting points and background of the design, implementation and tooling choices.

Utilizing existing functionality

To meet the hardware-independence requirement, we do not emulate specific TEE hardware with software based emulators, such as QEMU [35]. Instead we rely on existing technologies and the services offered by the mainstream OS in which Open-TEE is running rather than developing a new TEE OS to deploy the GP APIs in. In addition we reuse software from existing open source projects, such as the OpenSSL[4] crypto library and the GNU tool suite, thereby reducing the amount of time required to develop and test the Open-TEE framework.

[4]http://openssl.org

This also contributes towards meeting the ease of use requirement in that developers can easily set up Open-TEE and start developing TAs using a set of familiar tools, editors, Integrated Development Environments (IDEs), compilers and debuggers. For example, a developer utilizing Open-TEE can connect to a TA process with a cheap reliable software debugger such as GDB [19] for detailed debugging tasks like stepping through the code, inspecting variables and registers etc.

Development process

The intended user base for Open-TEE consists of seasoned developers. To ensure viability in such a demanding user base, we adopted a rigorous development process for Open-TEE so that the end result will be perceived as robust and usable. Open-TEE is developed as an open source project and as such there are a number of powerful tools that are freely available for this type of project. GitHub[5] is used for hosting the code and GerritHub[6] is used for performing peer-review of all code before it is submitted to the code base. In addition to the manual review process we leverage the power of Coverity[7] to perform in depth static analysis scans. This enforces secure coding practices and helps to find potential functional bugs that may have been missed during the manual code review. In addition, we have deployed a Continuous Integration (CI) server running Jenkins[8], which we have connected to GerritHub. Its main task is to perform a number of "smoke tests"[9] on the new patches. These tests ensure that the patches conform to the coding guidelines, build successfully and that the basic system is usable after the patches are applied.

Open-TEE in use

Being designed as an open source framework upon which to build and test features that will utilize a TEE, Open-TEE has been implemented to be as inconspicuous as possible. The complexity of the system is hidden from the users of Open-TEE. They are presented with an SDK that exposes the GP TEE Client and GP TEE Core APIs without being required to have a deep understanding of how the overall

[5] http://github.com/
[6] http://gerrithub.io/
[7] https://scan.coverity.com/projects/3441
[8] http://jenkins-ci.org/
[9] A suite of tests intended to ensure that the basic functionality of a system are intact.

framework works, thereby allowing them to focus on the development of their own TAs. However, Open-TEE is already being extended by the community. The ongoing implementation of the GP TEE Trusted User Interface (TUI) [16] [10] specification is an example.

Android and Open-TEE

During the Open-TEE specification phase it was recognized that there may be a demand for Open-TEE on a mobile device, because in general, most of the TEE use cases are associated with mobile devices and therefore it can be expected that most of the developed TAs are designed for this domain. From a developer's perspective it would be valuable to test the application in the context of the device where it will be deployed, to ensure that e.g. communication from upper level through CA to TA is working correctly before deploying it to real hardware.

Android was a natural target on mobile platforms for many reasons: it is widely deployed, open source and closely resembles GNU/Linux environment. Although Android was not the primary target, the Android aspect greatly affected the design of Open-TEE and implementation decisions, because the delta between the two environments should stay as minimal as possible in order to reduce the porting effort. To expand the example in the first paragraph, developers can test out their Java Native Interface (JNI) implementation with Open-TEE.

In the course of writing this thesis the Open-TEE Android port was succesfully completed with minimum effort.

SGX and Open-TEE

SGX provides a TEE service through a set of CPU instructions which can be used for constructing an "enclav" to protect sensitive code and data. This has a number of subtle side-effects, e.g. the "enclavised" code cannot make any system calls or invoke services provided by the OS directly. In other words, the protected code cannot rely on any service framework. The code must be self-contained. But as discussed in Chapter 2, one of the benefits of having a standards compliant TEE is the possibility

[10] https://github.com/Open-TEE

of using existing services. Developers could rely on the service framework provided by a TEE for e.g. accessing a keystore rather than maintaining the keystore by itself.

Using SGX within Open-TEE has been a consideration from the beginning of the Open-TEE project. Utilizing SGX it should be possible to expand Open-TEE from a development aid into a fully functional TEE. In the course of writing this thesis, the efforts of hardening Open-TEE has been an ongoing effort, which has greatly affected Open-TEE design and implementation decisions.

For example, one of the reasons for separating the TA processes into an I/O and a logic thread is to enable the logic thread to be "enclavised" within SGX and offload the OS interaction to the I/O thread.

Fall back TEE

In general, securing Open-TEE is an intriguing topic, because modern OSs are equipped with various security mechanisms like Seccomp[11] and LXC Containers[12]. The future research work question might be formulated as: Could it be possible to combine various OS security mechanisms into one product and what level of trust could be achieved by using these existing techniques and technologies? The edge of this approach is that the enabler technology is in place and it only needs to be utilized. In addition experiments could be conducted to see if these mechanisms are deployable in existing platforms and devices.

GlobalPlatform call for review

During our implementation of Open-TEE, GlobalPlatform announced an updated version of the GP TEE Core API. The implementation started with GP TEE Core version 1.0 and was nearly complete when GP announced a public review of the subsequent version. Under review was a working draft 1.0.26 and it received comments from all interested parties. This opportunity was utilized and the review of the working draft C was created in GP format and submitted to GP for public review.

The review is based on the experience of implementing Open-TEE:

[11] http://man7.org/linux/man-pages/man2/seccomp.2.html
[12] https://help.ubuntu.com/lts/serverguide/lxc.html

- It raised the issue of some of the function descriptions. They may be difficult to understand or are ambiguous. For example the working draft function TEE_PopulateTransientObject description was ambiguous of how the function parameters should be handled e.g. should they be deep or shallow copied[13].
- Pointing out some of the unrealistic functional requirements. Some of the functionality might be questionable after evaluating the feasibility of implementing these requirements from the TEE vendor's point-of-view. An example of this is a static allocation of a cryptographic operation. The cryptographic operation is allocated with TEE_AllocateOperation function, which is the only function the cryptographic API subsection of the GP TEE Core API that can return the out of memory return code. Therefore if our implementation needs to be fully GP compliant, it must ensure that cryptographic operations are not failing due to allocation operations.
- Proposing new functionality. For example related to operation state; a new constant was proposed for improving the TEE Core API readibility and usability.
- Pointing out possible flaws in the working draft. For example Authenticated encryption state was incorrect after TEE_AEinit function, which would block subsequent calls e.g. TEE_AEUpdate.
- Suggested re-ordering the document layout with an aim to improve readability of the document.

GP reviewed the proposals and provided feedback of the submitted review. They made multiple changes on the suggestions in the review. For example the TEE_PopulateTransientObject function description wording was improved. Because the subsequent version of GP TEE Core API was a minor release, they did not accept any major changes. They were strict about not breaking the backward compatibility and therefore a couple of points from review are considered for the next major version.

GP Trusted User Interface (TUI)

TAs without the possibility to interact with the user severely limit the possible TEE/TA use cases. For example exchanging sensitive information with the user,

[13]Data from A is actually copied to B rather than a referencing it.

i.e. when the user is entering authorization credentials (username, password) or the TA may want to display a calculated One-Time code to the user to allow them to log into a service. In such cases there is a need to interact with the TEE without the possibility of the REE intercepting the sensitive content. GP recognized this need and they have extended TEE specisification with TUI API [16].

As mentioned, there is an ongoing open source project[14], which is implementing the GP TEE TUI specification. The project builds on top of Open-TEE by implementing an extension, which provides the TUI functionality for TAs. One benefit of having a TUI framework with Open-TEE is that the developers are able to develop and test their TUI based TAs on the different platforms where Open-TEE is supported.

[14]https://github.com/Open-TEE

4. EVALUATION

This chapter assesses the requirements from Chapter 3 and evaluates how well Open-TEE meets them. The requirements are being continually evaluated, even as this thesis is being written, due to the ever evolving nature of Open-TEE. It is an active open source project to which more features are being added and existing features refined based upon the feedback that is received.

4.1 Compliance

Every effort has been made to comply with the GP standard. Whenever this has not been feasible, due to time constraints or in the interest of providing a platform upon which to build, the deviation has been documented and a debug message is logged to inform the user of the non-compliance. The GP TEE Client API is fully implemented. The GP TEE Core API implementation has 100% function coverage, however, the algorithm coverage is currently 80% due to the use of existing libraries that do not support the remaining algorithms.

Information related to other implementations of the GP specification are scarce and most of the information related to it is proprietary and therefore it is not known if some of the commercial TEE implementations are fully GP compliant. If this were known, an idealistic validation of our implementation would be executing CAs and TAs we have created in an independently implemented TEE to compare the results.

A compliance test suite is commercially available from GP, however it is not freely available and needs to be purchased by non-affiliated members. Because Open-TEE is an open source project, it lacks the funds to purchase the use of this tool.

4.2 Hardware-Independence

By following the GP standard and not emulating any specific TEE hardware, Open-TEE is independent of TEE hardware. TAs developed with Open-TEE can be compiled to any target TEE hardware architecture. We have verified [13] that a non-trivial TA developed using Open-TEE (284 Lines of Code (LoC), 19 GP TEE Core API invocations (9 unique functions), 6 invokable TA commands) has been successfully compiled and run on a hardware TEE based on ARM TrustZone running the Trustonic <t-base environment [41].

Open-TEE can provide coverage reports to help highlight hot-spots in the code, generate call graphs etc. The GP TEE Core API includes memory management primitives and allows configuration parameters (such as `gpd.ta.dataSize` and `gpd.ta.stackSize`) to indicate how much heap and stack memory is available to a TA. A developer can use these parameters to configure Open-TEE to reflect the memory restrictions of a target hardware TEE environment.

However, as the actual TEE is potentially running a different environment than that offered by Open-TEE– possibly utilizing hardware based cryptographic accelerators, potentially having a different CPU, with different clock speed and throughput characteristics – it will result in different timing characteristics. In this sense, as with all virtual environments, Open-TEE cannot fully replace the actual hardware environment for the final stages of the development cycle. Instead developers using Open-TEE can gain confidence that the hardware-independent parts of their trusted applications have been optimally implemented by making judicious use of coverage reports and other generic analysis techniques. Any hardware-specific optimization, such as performance tuning, naturally needs to be done on the target hardware environment.

Open-TEE has been deployed and used on various development environments ranging from servers to desktops and laptops[1]. It has been tested on both ARM and x86 architectures. Open-TEE requires Linux but has been run successfully on virtual machines hosted on other OSs. Having chosen not to emulate existing hardware to create the framework helps to ensure that the TAs created using it are portable as it is harder to create machine dependent code.

[1]Lenovo X1 carbon, MacBook pro, Samsung XE303C12, running Linux

4.3 Footprints and Performance

To evaluate our performance we deployed Open-TEE on a desktop machine (Intel i7-2600 CPU with 8GB RAM) running 64-bit Ubuntu 14.04. All performance tests were run 40 times while the machine was under normal load e.g. having editors and browsers open.

4.3.1 Disk Consumption

Open-TEE is written in ANSI C with a total of 12423 lines[2] spread over 78 source and header files. Table 4.1 shows the total size of the framework and highlights two libraries from the framework that are of most interest to developers, being "libInternalApi.so" against which the TAs are linked and "libtee.so" against which CAs are linked. As is standard on operating systems that support shared libraries the "Text" section, containing the program's code, can be shared among the different processes that link against it. The "Data" and "BSS" respectively refer to the initialized and uninitialized data parts of the library that can be shared in a Copy-On-Write (COW) basis. As the table highlights the vast majority of the libraries' size can be shared, thus reducing the required footprint.

Table 4.1 *Binary sizes (bytes)*

	Text	**Data**	**BSS**	**overall**
libInternalApi.so	117448	2248	160	119856
libtee.so	18617	880	152	19649
Total Framework	224948	7760	1664	234372

4.3.2 The Memory Consumption

Determining the absolute memory usage of Open-TEE is not feasible, because the GP standard does not define limits for the maximum number of CA/TA connections. It is an implementation defined property and Open-TEE is implemented in a way that it depends on the running environment i.e. how many sockets can one process acquire. But for having an overall indication of Open-TEE memory consumption, the memory consumption is examined under three different scenarios. These are:

[2]gathered using sloccount: http://www.dwheeler.com/sloccount/

Open-TEE framework itself, Open-TEE framework with one TA and Open-TEE framework and multiple TAs. The following memory measurements were collected and formatted into tables:

- **Resident Set Size (RSS)** shows how much memory has been allocated for process, this includes all memory that a process shares with other processes. As such it is a very naive measurement of a processes memory impact.
- **Shared** is the memory that a process shares with other processes, i.e. through the use of shared libraries.
- **Private** is the memory that is private to a process and will be returned to the system when the process terminates, however, Copy-On-Write semantics after a process fork may complicate this calculation. The Private pages may actually be shared until one or the other of the processes tries to write to the page, at which time it will be given its own copy of the Private page.
- **Proportional Set Size (PSS)** is a realistic indicator of the actual memory footprint of a process. It is calculated as the sum of the Private memory used by a process and the average Shared memory use per process. E.g., if a process has 100KB of Private memory and 1000KB of memory shared with 10 processes, its impact on system memory is 200KB[3]. Taking the example of Launcher between runs 2 and 3 we see that while the RSS, Shared and Private memory usage stay constant the PSS decreases as more pages are shared with the new TA.

Following results were collected.

1. The first scenario is Open-TEE framework by itself. Table 4.2 shows the memory consumption of Manager and Launcher immediately after they have been initialized, i.e. before any TAs have been launched. Manager process memory footprint could be reduced by removing all possible TAs binaries from Open-TEE "TA"-folder, because Manager process stores each of the the TA binary's Universally Unique Identifier's (UUID's) and name to its own data structures to enable loading of the TA when requested by a CA[4] and

[3] 100KB + (1000KB / 10) = 200KB
[4] TA binary format is ELF and TA UUID is stored as a section into binary

binary name to its own data structures. During the memory measurement the "TA"-folder contained two TAs.

Table 4.2 *Open-TEE framework memory usage*

	RSS	**Shared**	**Private**	**PSS**
Manager	1024	764	260	305
Launcher	1624	1232	392	558

2. The next scenario is Open-TEE framework with one TA. CA/TA may affect Manager process memory usage by registering shared memory regions. Every region is registered in Manager process and it is done for controlling resource usage and termination. The CA/TA, which was used in this measurement, registers one shared memory region which is used for transferring data between CA/TA. Table 4.3 shows how the memory consumption increases when one TA is launched.

Table 4.3 *Open-TEE memory consumption with one TA*

	RSS	**Shared**	**Private**	**PSS**
Manager	1112	832	280	316
Launcher	1648	1548	100	397
Test TA1[5]	1072	932	140	308

3. The last scenario is with multiple TAs. Table 4.4 shows the situation when two TAs are running simultaneously. This measurement continues from previous scenario. The first TA is not unloaded from Manager, because it has been configured to be "kept alive"[6].

Table 4.4 *Open-TEE memory consumption with two TAs*

	RSS	**Shared**	**Private**	**PSS**
Manager	1116	832	284	319
Launcher	1648	1548	100	337
Test TA1	1072	944	128	245
Test TA2[7]	1236	1068	168	299

[5]ta_conn_test_app
[6]TA context shall be preserved, when there are no sessions connected to the TA
[7]example_digest_ta

Overall, it can be concluded that (a) the memory footprint of Open-TEE is low and (b) the extensive use of shared libraries implies that the marginal memory cost of launching a new TA is small, as shown by the PSS figures. It can be also observed that Manager and Launcher process memory consumption does not (c) significantly increase, rather it starts to plateau. Table 4.5 highlights the delta between different scenarios. The first TA launch causes a rather high memory consumption, because i.e. data structures are getting initialized when they are first used.

Table 4.5 *Memory consumption delta between different runs*

	RSS	**Shared**	**Private**	**PSS**
Manager	1024	764	260	305
Launcher	1624	1232	392	558
Manager	88	68	20	11
Launcher	24	316	-292	-161
Manager	4	0	4	3
Launcher	0	0	0	-60

4.3.3 Build and Run Performance

One of the driving requirements of Open-TEE is the need to have short build and deploy cycles to help reduce the overall development effort. Table 4.6 highlights that Open-TEE does not pose a significant overhead to the developer, taking an average of just 147 ms to perform an incremental build of a TA. The time required for an incremental build was comparable to that of a clean build, falling within the standard deviation of the former, this can be attributed to the source code being confined to a single C file. Comparative results are not available for deployed hardware-based development environments. However, considering that a full reset of the target device and the subsequent boot of its OS may be required before the CA can be launched, Open-TEE's performance is likely to be perceived as being superior.

Table 4.6 *Average build and execute times of a TA, including standard deviations*

	Time
Build	147 ms $\pm$ 10.95
Execute	430.5 μs $\pm$ 32.6

4.4 Ease of Use

Determing whether Open-TEE eases the burden of TA development, is particularly challenging because until now, TA development has been limited to a very small set of developer's and further more ease of use is a loose definition, which is open to interpretation. It depends on a developers current environment and how she/he defines easy. A user study was conducted, for collecting a variety of user experiences and feedback about Open-TEE usage. The results of the study were used for evaluating, if Open-TEE meets the "Ease of use" requirement in Chapter 3.

4.4.1 User Study

The user study was conducted in conjunction with the Intel Collaborative Research Institute for Secure Computing (ICRI-SC) Helsinki Team[8].

Participants

Our user study was publicly available[9]. The user study itself did not need any special background skills or knowledge. Anyone who was interested in this topic was invited to participate in our study, but it was mainly aimed toward developers who were acquainted with TEEs and have developed TAs.

As discussed above, suitable participants for user study are scattered, but fortunately, we were able to recruit several experienced TA developers from multiple organizations to participate in a user study. Fourteen people participated in the study. All had prior software experience (between 3 and 33 years, $M = 13$, $SD = 8.2$). Eleven had prior experience developing/debugging TAs (between $\frac{1}{2}$ and 15 years, $M = 5.1$, $SD = 4.2$).

Materials

The standard System Usability Scale (SUS) [8, 7] questionnaire was used to elicit the participants' estimates of the ease of use in developing TAs. We used a pre-

[8]Intel Collaborative Research Institute for Secure Computing (ICRI-SC) `http://www.icri-sc.org/icri-sc/institute/`

[9]The user study `http://open-tee.github.io/userstudy/`

Table 4.7 *Mean, standard deviation and median for the pre- and post-study SUS scores*

	Mean	**Std.dev.**	**Median**
Pre-study SUS	51.82	24.70	62.50
Post-study SUS	74.09	15.01	77.50
Post-study SUS (all participants)	69.92	18.09	68.75

study A and a post-study questionnaire B. In addition to demographic information, the pre-study questionnaire included free-form questions about the current software development environment (if any) they use for TA development. It also contained a SUS questionnaire which the participants were asked to complete with their current TA development environment in mind. This was completed only by those participants who had prior TA development experience.

The material for the user study[10] task was a sample CA/TA pair, provided as part of the Open-TEE source tree. A software flaw had been introduced to the TA, which, when executed, would result in a segmentation fault and subsequent premature termination of the TA. The CA was free of error and was only used to interact with the TA running in Open-TEE.

The post-study questionnaire consisted of a SUS form which the participants were asked to complete with Open-TEE in mind. The questionnaire also had open ended questions about specific difficulties they face in TA development.

Procedure

The user study was conducted in three steps. In the first step, participants were first asked to complete the pre-study questionnaire. The pre-study questionnaire purpose was to collect background information on the participants. After this they were pointed to a web page containing brief instructions on how to install and use Open-TEE. In the second step, once the participants completed the tutorial they were told about the flawed TA. They were tasked to identify the reason for the TA malfunction using Open-TEE and correct the flaw in the TA. Finally, in the third step, after the participants had completed the debugging exercise, they were asked to complete the post-study questionnaire.

[10]The user study materials can be found at `http://open-tee.github.io/userstudy/`

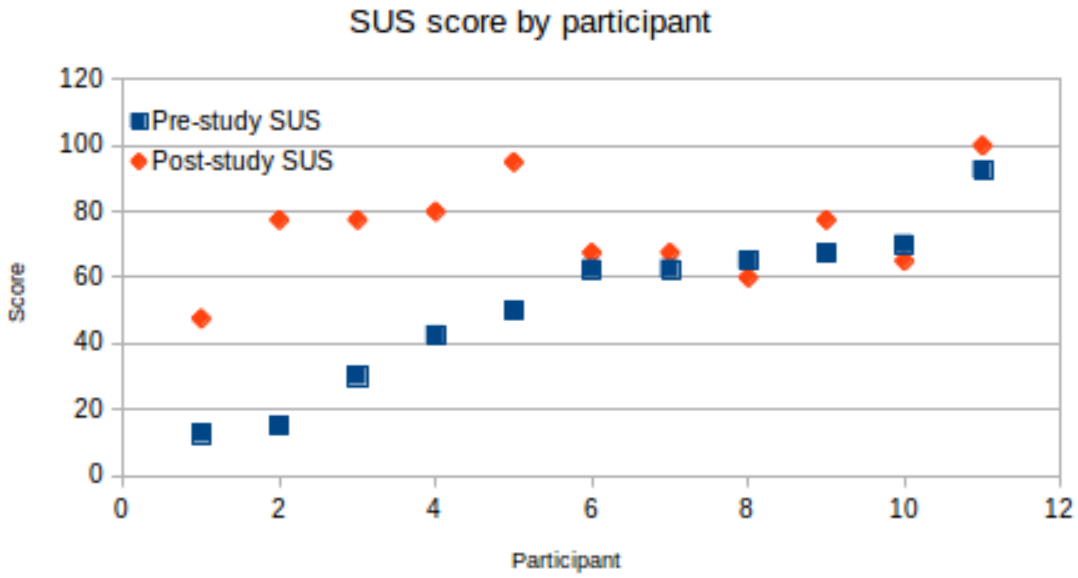

Figure 4.1 *Pre- and post-study SUS score (for participants with prior TA development experience)*

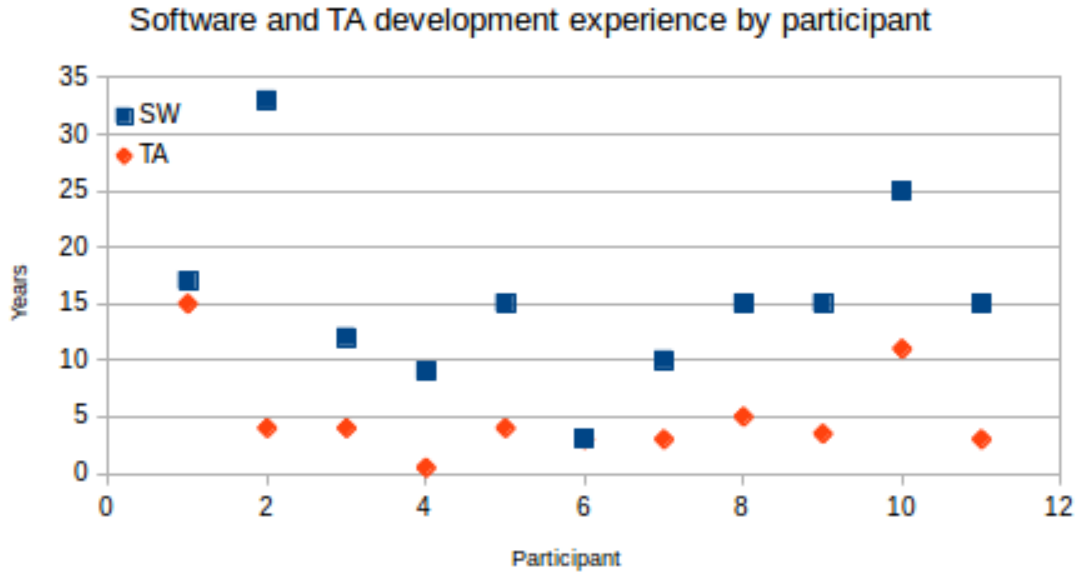

Figure 4.2 *General software and TA development experience by participant (for participants with prior TA development experience)*

4.4.2 User Study Result

The mean, standard deviation and median of the SUS scores for all participants, including those without prior TA development experience are shown in Table 4.7. With both sets of participants, the post-study questionnaire yields a mean score above 68, which is considered the threshold value for an above average SUS score, indicating an acceptable level of usability in Open-TEE.

Figure 4.1 shows the scores reported both before and after the use of Open-TEE by participants with prior TA development experience. Nine out of the eleven participants (82%) rated Open-TEE higher than the development environment they are using currently. This suggests that the perceived usability of Open-TEE is higher than that of the current tools used by the experienced TA developers. In five cases (46 %), the difference in SUS scores was 35 or more. In the remaining six cases, the difference in SUS scores was 10 or less. A Wilcoxon signed-rank test showed that the difference in SUS scores is statistically significant ($z = -2.50$, $p < .05$, $r = -0.53$).

The difference in SUS scores divides the participants into two distinct groups. The five participants for whom the difference was 35 or more had SUS scores below 60 in their pre-study questionnaire. The remaining six for whom the difference was 10 or less had pre-study SUS scores over 60. A natural question is whether we can discern any other difference between the two groups that might explain the difference in SUS scores. One possible explanation was that experienced software developers were comfortable with their current tools and hence did not perceive Open-TEE as being easier to use. If this explanation is correct then one can hypothesize that developers with many years of general software or TA development experience will rate their current development tools higher than their counterparts with fewer years of experience would. However, a Spearman's rho correlation test indicated no significant correlation between the years of general software development experience and the SUS score in the pre-study questionnaire ($r_s = -.042$, $p > .05$), nor between the years of TA development experience, and the SUS score in the pre-study questionnaire ($r_s = -.204$, $p > .05$).

Figure 4.2 shows the software development experience (both general and TA) reported by each participant whose SUS scores are shown in Figure 4.1.

A majority of the experienced TA developers (7 out of 11, 64 %), reported using hardware tools for debugging TAs under development. Four (36 %) used Lauter-

bach[11] hardware assisted debug tools. Three (27 %) used other development boards such as Arndale[12], Fido[13] or DS-5[14] or actual mobile devices. Participant responses highlighted different types of difficulties in debugging TAs using only hardware:

- workflow slowdown due to the need to (cross) compile, load and execute TAs on separate hardware (*"slow execution (flash, download, reboot, run)"*, *"debugging TA is slow, you need to cross compile and push binary into target hardware"*),
- problems due to the hardware itself being under development and hence exhibiting flaws, (*"TEE itself might not work without problems, because some change have been made"*),
- inconvenience caused by the restricted access to prototype hardware *"Main difficulty is that you need development hardware, which is problematic when working outside the office."*).

Six participants (55 %) reported that their current development environment does not support interactive debugging. But even the rest, who used tools like Lauterbach tracing, reported that they found it easier to resort to print tracing, whenever they needed to examine values of TA variables.

After having used Open-TEE, several participants commented *"debugging is easy"* or *"debugging is fast"* in the post-study questionnaire. One participant characterized how Open-TEE could be integrated into his existing workflow before cross compiling to target hardware: *"[Open-TEE] complements nicely my previous SDE - first preliminary testing with Open-TEE & `gdb` & `OT_LOG(..)`, and only after that ARM cross compiler & FVP emulation"*. The dominant suggestion for improvement was a desire to see more extensive documentation for Open-TEE.

4.4.3 Ease of Use Conclusion

Our user study was conducted for the purpose of determing if Open-TEE meets the ease of use requirement in Chapter 3. The study was successfully conducted. Given

[11] `http://www.lauterbach.com/`
[12] `www.arndaleboard.org/`
[13] `http://www.liewenthal.ee/projects/fido/`
[14] `http://ds.arm.com/`

the sample size, the results should be taken as indicative rather than definitive. However, it is reasonable to conclude that Open-TEE has the potential to improve the ease of use of developing TEE applications. We also collected the improvement ideas for Open-TEE.

5. RELATED WORK

Ekberg et al. [14] list several reasons for the underutilization of TEEs in devices: e.g., lack of standard APIs and easily available SDKs and lack of trust between the different stakeholders, with OEMs being unwilling to open up their security environments to third parties. This section reviews a number of initiatives that have been undertaken to address some of these issues and compare these efforts to Open-TEE.

On-board Credentials (ObC) [24] was one of the first attempts to address the problem of opening the TEE to third party developers by challenging the prevailing opinion that a credential system must be centralized and closed. ObC has been implemented and deployed in experiments with New York's public transport ticket sales. ObC predates many standardization efforts and as such defines a proprietary mechanism by which to enable the CA/TA communication and synchronization while leveraging the TrustZone architecture to enforce the security. On the other hand Open-TEE work aims to promote standards adoption in order to proliferate TEE research and deployment.

Muthu [30] analyzes extending QEMU to support TrustZone, the feasibility of such a solution, and tries to determine if it would be beneficial to the developer community. In his other work Muthu (et al.) [31] addresses a similar problem by considering the feasibility and benefits of emulating TrustZone in Android. Winter et al. [42] goes one step further and implements a TrustZone emulator as an open source project. However, the location of emulator source code is not included in the paper and therefore is not to be found. Open-TEE addresses the issue of virtualizing the TEE, however, in contrast Open-TEE is not tied to the emulation of a specific TEE implementation. One issue with developing an emulator for the TEE is that it still lacks an operating system to run. Section 2.4 highlights the lack of a standardized OS even among the different TrustZone implementations.

To this end there have been a number of efforts to create an OS that is suitable to be

deployed in TrustZone [33] [27] [40]. All of these are open source solutions which are released under various licenses (see Table 5.1). In addition to providing an operating system for the TEE both OP-TEE [27] and T6 [40] choose to rely on GP as their RPC mechanism between the REE and TEE. Trusted Little Kernel (TLK) [33] on the other hand chooses to provide a proprietary communication mechanism.

Sierraware's Open Virtualization [36] provides a dual-licensed OS implementation[1] that also supports the GP standards. The commercial products (sierraVisor, SierraTEE) provide extended functionality which is not General Public License (GPL) and there is no requirement for any changes to be made publicly available by the license holders as is required with their open source offering. The downside of the Open Virtualization open source product is that it offers only a subset of the features, compared to the commercial product and it is licensed under GPL. Open-TEE is licensed under Apache-V2 giving users the flexibility of an open source license without the strict copy left requirements.

Trustonic's <t-dev developers program [41] was created to support Trustonic partners who have deployed the <t-base TEE implementation. This program provides an SDK, tools and consulting with the aim of easing the development and testing of TEE applications in deployed hardware solutions.

Table 5.1 compares available solutions to Open-TEE. "Compliance" column indicates, if the solution is implemented according to GP TEE specification. If the solution requires specific hardware, information about this is collected into the "HW-independence" column. Table 5.1 uses question marks, if credible information cannot be found.

Regarding to GlobalPlatform compliance, a compliance test suite is commercially available from GP, thus, it is not freely available and needs to be purchased by non-affiliated members. In fact, GP test suit is the only test suite for compliance testing at the time of writing this thesis. Due to lack of a uniform method of confirming GP compliance, there is no coherent information about how well existing TEE hardware conform to the GP specifications.

All of the OS based solutions have to be ported to support the various hardware environments, increasing the effort of maintaining the OS and reducing the user's available options. Many of them also require that the hardware be configured in a

[1] GPL [20], proprietary

Table 5.1 *Comparison of available alternatives to Open-TEE*

	Compliance	**HW-independence**	**License**
Open-TEE	yes	yes	Apache-V2
Open Virtualization [36]	yes	no	proprietary, GPL
OP-TEE [27]	yes	no	BSD-2,BSD-3
T6 [40]	?	no	?
TLK [33]	no	no	MIT,FreeBSD
TrustZone Emulator [42]	?	no	?

developer mode, without this setting it is generally not possible to deploy custom software to the TEE, for obvious reasons, further restricting the developer's options. Open-TEE in contrast does not have this hardware dependency, thus enabling the users to start developing with the framework once they have cloned the repositories[2]. Based on the references listed above, it can be concluded that no other project fills the niche of a fast prototyping SDK framework that is described in this thesis.

[2] `http://open-tee.github.io/`

6. CONCLUSION

This thesis has demonstrated that Open-TEE meets the objective of an easy-to-use, hardware-independent software framework that allows developers to write and debug GP-compliant TEE applications. Open-TEE has been made deliberately open source under Apache-V2 license [37]. The Apache license was selected because it is a recognized open source license and it provides additional flexibility for those wishing to use the framework. All third party components have been carefully selected – the project has used only components that have been properly licensed and do not set any restrictions for future use. This has made it possible for a community to contribute to Open-TEE easily. Currently a number of extensions are being worked on including support for other GP APIs like GP TEE TUI API and supporting TEE Client API bindings in Java (for Android applications).

Although the sample in the user study is small, participants were drawn from several different organizations with track records of TA development. Thus it could be said with confidence, that the results of this user study are valid. It is very difficult at this time to conduct larger-scale user study of TA development because the community of TA developers is tiny. It could be said that expanding the size of the TA developer base is the motivation for Open-TEE in the first place.

Open-TEE was initially intended to be a developer tool. However, an alternative use has become evident in our discussions with service providers. Although use of TEEs can improve the security and usability of their service, not all their clients may have TEE-equipped devices. Yet the service provider would like to present a consistent user experience for their entire client base. A possible approach for them is to ship their application (CA and TA) with Open-TEE and arrange for the CA to use Open-TEE if it cannot detect a real hardware TEE on the device. This would allow the service provider to have a common provisioning mechanism and offer a consistent user experience for all their clients. However, once Open-TEE has been cast as a potential fall-back TEE in this manner, there emerges the need to address

the question of how it would be best to isolate it from the REE in the absence of any hardware support.

Reiterating that Open-TEE is not intended to emulate any specific TEE hardware. Open-TEE meets its goal of guaranteeing that trusted applications developed using it will compile and run on any GP-compliant TEE hardware. Hardware-specific aspects, such as performance tuning are outside the scope of Open-TEE.

During the course of writing this thesis, GlobalPlatform announced a public review of the next version working draft of GP TEE Core API. Based on Open-TEE implementation experience detailed feedback was provided to GlobalPlatform in response to their solicitation of public comments. Feedback included errors and ambiguities in the specifications. Several items in our feedback have been addressed in the released version 1.1.

The hope in writing this thesis is to make the research community aware of Open-TEE and encourage researchers to use it and contribute to its development. It is also believed that organizations and developers who already develop TA applications will benefit from incorporating Open-TEE into their development process.

BIBLIOGRAPHY

[1] Android Open Source Project, "Managing your app's memory," https://developer.android.com/training/articles/memory.html.

[2] Apple, "iOS security," https://www.apple.com/ca/iphone/business/docs/iOS_Security_Feb14.pdf.

[3] ARM, "Technical reference manual: ARM 1176jzf-s (trustzone-enabled processor)," http://www.arm.com/pdfs/DDI0301D_arm1176jzfs_r0p2_trm.pdf.

[4] ARM, "ARM security technology — Building a secure system using TrustZone technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html, April 2009.

[5] J. Azema and G. Fayad, "M-Shield mobile security technology," 2008, TI White paper. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[6] D. Balfanz and E. W. Felten, "Hand-held computers can be better smart cards," in *Proceedings of the 8th USENIX Security Symposium, Washington, D.C., August 23-26, 1999*, 1999. [Online]. Available: https://www.usenix.org/conference/8th-usenix-security-symposium/hand-held-computers-can-be-better-smart-cards

[7] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability," *International Journal of Human-Computer Interaction*, pp. 574–594, 2008, http://dx.doi.org/10.1080%2F10447310802205776.

[8] J. Brooke, *Usability evaluation in industry*. Taylor & Francis, London, 1996, ch. SUS: A "quick and dirty" usability scale, pp. 189–194.

[9] Y. Cheng, X. Ding, and R. Deng, "Appshield: Protecting applications against untrusted operating system," *Singaport Management University Technical Report, SMU-SIS-13*, vol. 101, 2013.

[10] Common Vulnerabilities and Exposures (CVE), "Cve-2015-3456," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456.

[11] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 secure coprocessor,"

IEEE Computer, vol. 34, no. 10, pp. 57–66, 2001. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/2.955100

[12] J.-E. Ekberg, "Securing software architectures for trusted processor environments," Doctoral dissertation, Aalto University, May 2013, http://urn.fi/URN:ISBN:978-952-60-3632-8.

[13] J.-E. Ekberg, "Personal communication," 2015, Trustonic.

[14] J. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 29–37, 2014. [Online]. Available: http://dx.doi.org/10.1109/MSP.2014.38

[15] GlobalPlatform, "About." http://www.globalplatform.org/aboutus.

[16] GlobalPlatform, "Device specifications for trusted execution environment." http://www.globalplatform.org/specificationsdevice.asp.

[17] GlobalPlatform, "Home page." http://www.globalplatform.org.

[18] GlobalPlatform, "TEE System Architecture," http://www.globalplatform.org/specificationsdevice.asp.

[19] GNU, "GDB: The GNU project debugger," http://www.gnu.org/software/gdb/.

[20] GNU, "General public license," https://gnu.org/licenses/gpl.html.

[21] Intel, "Intel software guard extensions (intel sgx)," https://software.intel.com/en-us/intel-isa-extensions#pid-19539-1495.

[22] Intel, "SEP driver," https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/staging/sep?id=refs/tags/v3.14.32.

[23] Intel, "Software guard extensions programming reference," https://software.intel.com/sites/default/files/329298-001.pdf.

[24] K. Kostiainen, J. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, 2009, pp. 104–115. [Online]. Available: http://doi.acm.org/10.1145/1533057.1533074

[25] K. Kostiainen, E. Reshetova, J.-E. Ekberg, and N. Asokan, "Old, new, borrowed, blue–: a perspective on the evolution of mobile platform security architectures," in *Proceedings of the first ACM conference on Data and application security and privacy*. ACM, 2011, pp. 13–24.

[26] M. Leno, "Senate bill 962, leno. smartphones." http://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201320140SB962.

[27] Linaro, "OP-TEE," https://wiki.linaro.org/WorkingGroups/Security/OP-TEE.

[28] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for TCB minimization," in *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, 2008, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352625

[29] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[30] A. Muthu, "Emulating trust zone feature in android emulator by extending qemu," Master's thesis, KTH Royal Institute of Technology, 2013.

[31] A. Muthu, R. Rahmani, and D. Rajaram, "Emulating trust zone in android emulator with secure channeling," *International Journal of Computer Science Issues*, vol. 10, no. 5, pp. 40–51, 2013.

[32] R. Needham and A. Herbert, "The cambridge cap computer and its operating system," 1982.

[33] NVIDIA, "Trusted little kernel (tlk)," http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=summary.

[34] Official California Legislative Information, "Senate bill no. 962," http://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201320140SB962.

[35] QEMU, "Open source processor emulator," http://wiki.qemu.org/Main_Page.

[36] Sierraware, "Open virtualization's SierraVisor and SierraTEE," http://www.openvirtualization.org/.

[37] The Apache Software Foundation, "Apache license, version 2.0," http://www.apache.org/licenses/LICENSE-2.0.

[38] The Chromium Projects, "Multi-process architecture," http://www.chromium.org/developers/design-documents/multi-process-architecture.

[39] "Trusted Platform Module (TPM) Specifications," https://www.trustedcomputinggroup.org/specs/TPM/.

[40] TrustKernel, "T6," http://trustkernel.org/.

[41] Trustonic, "<t-dev developer program," https://www.trustonic.com/products-services/developer-program/.

[42] J. Winter, P. Wiegele, M. Pirker, and R. Tögl, "A flexible software development and emulation framework for ARM TrustZone," in *Trusted Systems - Third International Conference, INTRUST 2011, Beijing, China, November 27-29, 2011, Revised Selected Papers*, 2011, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32298-3_1

A. PRE STUDY QUESTIONNAIRE

Thank you for agreeing to participate in our study. Our goal is to understand the difficulties encountered by developers of Trusted Applications (TAs) for Trusted Execution Environments (TEEs). First we would like to learn about yourself and the TA development environment you customarily use.

1. Name:

2. E-mail address:

3. How long have you been developing TAs?

4. How long have you been doing software development in general?

5. What software development environment do you use for developing TAs?

   If you use a particular IDE, give its name (e.g., "Eclipse")
   Otherwise, give names of tools you use for developing TAs (e.g, "gcc, gdb, make, lint)

6. Do you use any hardware tools for developing/debugging TAs? If so, please list them.

7. Compared to ordinary software development, what challenges/difficulties do you face in developing TAs? List up to 5 items in descending order of priority (most important first)

The following set of statements is a standard form used to estimate usability of a system. The "system" in this case is the software development environment (SDE) you use (i.e., your response to Question #5 in the previous page).

Think about this SDE you have been using so far to develop your TAs and for each of the following statements, mark one box that best describes your experience with it so far. The scale is increasing agreement from left to right:
1 = "Strongly Disagree"
5 = "Strongly Agree

		1	2	3	4	5
		Strongly Disagree				**Strongly Agree**
1.	I think that I would like to use this SDE frequently.	●	●	●	●	●
2.	I found this SDE unnecessarily complex.	●	●	●	●	●
3.	I thought this SDE was easy to use.	●	●	●	●	●
4.	I think that I would need assistance to be able to use this SDE.	●	●	●	●	●
5.	I found the various functions in this SDE were well integrated.	●	●	●	●	●
6.	I thought there was too much inconsistency in this SDE.	●	●	●	●	●
7.	I would imagine that most people would learn to use this SDE very quickly.	●	●	●	●	●
8.	I found the SDE very cumbersome to use.	●	●	●	●	●
9.	I felt very confident using the SDE.	●	●	●	●	●
10.	I needed to learn a lot of things before I could get going with this SDE.	●	●	●	●	●

B. POST STUDY QUESTIONNAIRE

Thank you for using Open-TEE. Our goal in this study is to understand to what extent Open-TEE addresses the difficulties encountered by developers of Trusted Applications (TAs) for Trusted Execution Environments (TEEs).

1. Name:

2. E-mail address:

The following set of statements is a standard form used to estimate usability of a system. The "system" in this case is Open-TEE

For each of the following statements, mark one box that best describes your experience with Open-TEE so far.
The scale is increasing agreement from left to right:
1 = "Strongly Disagree"
5 = "Strongly Agree

		1	2	3	4	5
		Strongly Disagree				**Strongly Agree**
1.	I think that I would like to use Open-TEE frequently.	☐	☐	☐	☐	☐
2.	I found Open-TEE unnecessarily complex.	☐	☐	☐	☐	☐
3.	I thought Open-TEE was easy to use.	☐	☐	☐	☐	☐
4.	I think that I would need assistance to be able to use this Open-TEE.	☐	☐	☐	☐	☐
5.	I found the various functions in Open-TEE were well integrated.	☐	☐	☐	☐	☐
6.	I thought there was too much inconsistency in this Open-TEE.	☐	☐	☐	☐	☐
7.	I would imagine that most people would learn to use this Open-TEE very quickly.	☐	☐	☐	☐	☐
8.	I found Open-TEE very cumbersome to use.	☐	☐	☐	☐	☐
9.	I felt very confident using the Open-TEE.	☐	☐	☐	☐	☐
10.	I needed to learn a lot of things before I could get going with Open-TEE.	☐	☐	☐	☐	☐

1) Name up to three positive things about your experience using Open-TEE.

2) Name up to three negative things about your experience using Open-TEE.

3) Do you have any suggestions for improving Open-TEE to make it more useful in easing TA development?

4) Tell us about your current practices in developing and debugging TAs using your current software development environment (SDE):

   a) Does your SDE allow interactive debugging?

   b) Suppose you need to check the values of a variable at different points during TA execution. How do you do this in your current SDE?

   c) What does your SDE cost? (e.g., price of a single developer license / hardware unit)

C. GP TEE CORE API V1.1.0.26 PUBLIC REVIEW

07 April 2014

GlobalPlatform Device Technology
TEE Internal Core API Specification
Version 1.1.0.26 – Public Review
Please send to: tee-int-core-api-review@globalplatform.org

COMPANY: Open-TEE project (https://github.com/Open-TEE)
Name: Tanel Dettenborn, Brian McGillion
Date: 22.05.2014

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
1	80	4.11	T	New function. If TA maximum memory is defined, there should be a function for querying how much free memory there is available for a TA. When a TA knows the available memory, it could determine next available action.	Add new function: - uint32_t TEE_GetMemoryAvailable(); Return value is available memory as bytes.
2	89	5.1 § 2 – line 2	T	Unnecessary complexity. Persistent object 0-byte identifier length is a special case (perspective of TEE implementation) and gaining benefits over 0-byte length is arguable.	Persistent object identifier length is from 1 to 64 bytes.
3	90	5.1 § 7 – line 4	T	Unnecessary functionality. When TEE_ERROR_CORRUPT_OBJECT(_2) return code is returned; any attempt at continuing to use the corrupted object handle will cause a panic. This should not happen if the closed object handle is set to null. Use of a null handle might cause panic, but that panic is not related to corrupted handle. Related also #21.	Remove sentence “subsequent use of the handle SHALL cause a panic.”
4	90	5.1	T	Useful functionality. Persistent object might not be corrupt, when TEE_ERROR_CORRUPT_OBJECT is returned. For example the object handle might get corrupted yet the persistent object in storage is unaffected. The same could also occur with a reading error.	Introduce new return code TEE_ERROR_CORRUPT_HANDLE. Return code is only used with persistent object and only if persistent object data at storage is not corrupt. In such a case the handle is not closed nor persistent object deleted from storage. It would then be at the callers discretion as to how best to proceed, e.g. the user could close the object and re-open it. A new return code could be defined, for example, TEE_GetObjectInfo function (#6).
5	96	5.5.1 § 2	T	Ambiguous definition. The *KeySize* variable needs a more precise definition. If object size is terminated by object attributes, the size can be greater than *maxKeySize* variable. For example RSA-key consists of multiple components and lets assume that RSA modulo is equal to *maxKeySize*.	Guessing. *KeySize* variable is representing current key strength and is determined for example by RSA modulo length? Add clarification: *keySize*: Representing current key strength.

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
				Which leads inevitably to *keySize* > *maxKeySize*.	
6	96	5.5.1 § 3	E	Deprecated naming. Variable *maxKeySize* name is not self explaining.	Add further clarification for example: (bullet) *maxKeySize*: Representing object cryptographic key maximal strength.
7	99	5.5.3 § 1	T	Uniform and useful functionality. TEE_GetObjectValueAttribute allows for parameters *a* and *b* *to* be null, TEE_GetObjectBufferAttribute *buffer* and *size* should also be allowed to be null.	It is a security risk, if TEE_GetObjectBufferAttribute *size* parameter is null and *buffer* parameter is not (possibility of buffer overflow). If the caller wishes to check whether the object contains an attribute both buffer and size could be null. If *buffer* parameter is null and the size is not, the *size* could contain the required length of the buffer on return, to allow the caller to allocate the required space for the buffer (if attributes are found). Also useful for just checking the attribute length.
8	99	5.5.3	T	Ambiguous definition. If the object is not initialized, the TA panics. This panic reason overlaps the return code TEE_ERROR_ITEM_NOT_FOUND, because if the object is not initialized then nothing is not be found either.	If object is not initialized, function could return code TEE_ERROR_ITEM_NOT_FOUND. Thus improving usability.
9	100	5.5.4	T	Ambiguous definition. Apply same conclusion as point #8.	Apply same proposition as point #8.
10	102	5.6.1	T	General functionality. Apply same conclusion as point #12.	Apply same proposition as point #12.
11	101	5.5.5	T	Unrealistic requirements. TEE_CloseObject is a void function, it is unreasonable to assume that the close operation can always complete without incident. This leads to the persistent object always being in “closed” state. Most close operations also flush any remaining buffers but this function as defined would only allow for the freeing of the object handle from the TA's memory.	Function should have return values: -TEE_SUCCESS: In case of success. -TEE_ERROR_CORRUPT_OBJECT: If the persistent object is corrupt.
12	102	5.6	T	General functionality. Requirement for allocating all resources in TEE_AllocateTransientObject function is setting up unfeasible and resource wasting transient objects. - Infeasible: Apparently you should allocate all your resources here and in effect this means that it should also,	Transient object allocation should be divided. TEE_AllocateTransientObject function should allocate a new object handler and meta structures for the expected attributes. TEE_PopulateTransientObject and

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
				for example, allocate all temporary buffers which would be needed by other functions. - Infeasible: You cannot use third party solution. Because this solution could use, for example, temporary buffers. - Resource wasting: If an object is not initialized, it is only wasting resources. - Resource wasting: If object is initialized with weaker key and this generally means less space for key, but object is allocated according to it *maxKeySize* → Resource waste - Resource wasting: Attribute size can not be predicted exactly (can not be assumed it is used in "standardized" way). For example if one is calculating RSA modulo from different sized prime numbers. One of the used primes could be smaller than that other but both are allocated according to *maxKeySize*. - Generally: The benefits gained by have a single allocation point and the complexity of implementation are questionable. There is no guarantee that the operation will not fail due to lack of resources. - Generally: This approach is inflexible going forward. The allocating function will become large and overly complex, and it still requires special functionality in other functions	TEE_GenerateKey functions should allocate reference attribute buffers and populate them according to the received parameters or generated key. TEE_PopulateTransientObject and TEE_GenerateKey functions need to have a new return code TEE_ERROR_OUT_OF_MEMORY added to support this added functionality. Also it should be noted that TEE_CopyObjectAttributes will also need to be able to return TEE_ERROR_OUT_OF_MEMORY if the situation arises. Note: This is only an overview of the problem. If you required, we can draft a more detailed version that is in line with the overall specification.
13	102	5.6	T	Useful functionality. See first #14. This point is only valid if TEE_PopulateTransientObject attributes are copied.	Proposing new object handle flag: TEE_HANDLE_FLAG_REFERENCE. Flag can only be set to transient object and only prior initialization. If flag is set, the object attributes are not copied during a call to TEE_PopulateTransientObject, instead the ownership of the attributres is transferred to the transient object. This prevents a double allocation for the same object. Of course attribute length should be consistent with object maxKeysize.
14	107	5.6.4	T	Undefined behavior. TEE_PopulateTransientObject does not define how parameters should be handled. Should parameters in *attrs*-arrays be deep copied (including the reference attribute buffer) or should a shallow reference be created? If taking only a reference, should the ownership be	In the function specification precise language e.g. "deep copy" / "transfer ownership" should be used.

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
				transferred?	
15	107	6.6.4	T	Unnecessary functionality. If parameter *attr* contains an unexpected parameter, this will force a panic.	Remove panic reason and allow extra attributes in the attribute array. Additional benefit of proposition is that the caller could use the same attribute array in different populate/generate calls. TEE_PopulateTransientObject could then only copy necessary parameters thus aiding usability and efficiency.
16	111	5.6.6	T	General functionality. Apply same conclusion as point #12.	If destination object is consistent with source object, destination object attributes are copied (reference attribute buffers are allocated prior to copy). The function will require the possibility to return TEE_ERROR_OUT_OF_MEMORY.
17	111	5.6.6	E	Clarification. The specification should point out the curiosity that the destination object key strength should only be equal to or less than source object. Never greater.	Note or warning that copied object is weaker.
18	113	5.6.7	T	General functionality. Apply same conclusion as point #12.	Apply same proposition as point #12.
19	118	5.7.2	E (T)	Unnecessary complexity. See also #2. What could be practical use case for 0-byte ID length?	Apply same proposition as point #2.
20	124	5.7.5	E (T)	Unnecessary complexity. Apply same conclusion as point #19.	Apply same proposition as point #19.
21	125	5.8	T	Useful error. It is not known beforehand what is the size requirement when allocating an enumerator. Because enumeration is not started during the allocation. It would be more appropriate for a TEE implementation if you can allocate memory in TEE_StartPersistentObjectEnumeration. This could allow optimization of the TEE implementation.	Should allow TEE_ERROR_OUT_OF_MEMORY in TEE_StartPersistentObjectEnumerationfunction. Function specification should point out that extra resource allocation is not recommended.
22	125	5.8	T	New functionality. Removing a persistent object from storage is wasting resources. To remove a persistent object user must allocate/open persistent object and then remove persistent object. Removing object from storage in a more efficiency way should be available, if for example TA is cleaning up its storage.	New function: - TEE_DeletePersistentObject(void *objectID, uint32_t objectIDLen); Function to remove persistent object from storage. Prior to deletion this function MUST check object access rights and MUST be “granted” meta

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
					access right. - Return TEE_SUCCESS if meta access right was granted and object deleted from storage. - Return TEE_ERROR_ACCESS_CONFLICT if access right conflict was detected. - TEE_ERROR_ITEM_NOT_FOUND if persistent object not found.
23	132	5.9.2	T	Useful requirement. Function specifies that a write operation should be atomic and that is a justifiable requirement. But it is hard to leverage in a TEE implementation if arbitrary block writing is not required to be atomic.	Define block size that must be atomically written to storage. For example TEE must guarantee that up to 4k block is written atomically and blocks over 4k is written non atomically. Non-atomically written data must be monitored to detect if the write operation failed and return code TEE_ERROR_OBJECT_CORRUPTED.
24	137	6.1	T	Useful constant. It could be useful if there would be a constant for operation state. Defining constant would make specification more readable. TEE implementation does not have to be aware, is 1 or 0 active state.	Make a subsection 6.2 Constant to chapter 6.1 and define TEE_OPERATION_STATE_ACTIVE = 1, TEE_OPERATION_STATE_INITIAL = 0 Change explanation at page 146, chapter 6.2.4 bulletin 7: operationState: Filled with operation state
25	138	6.1.3	T	Possible error. The size of *KeyInformation*-array is hard coded to one, but if an algorithm requires multiple keys, for example AES-XTS the array would not be large enough to support this requirement.	Hard code array size according to its biggest size. The biggest size is determined by which algorithm requires most keys. Currently this is the two key requirement of AES-XTS.
26	139	6.2	T	Unfeasible requirements. Apply same conclusion as point #12.	Operation allocation should be divided. TEE_AllocateOperation should allocate a new operation handler and meta structures for expected attributes. TEE_SetOperationKey(2) should allocate reference attribute buffers and copy attributes according to received parameters. The TEE_SetOperationKey(2) functions would require the ability to return TEE_ERROR_OUT_OF_MEMORY if such a situation is encountered.. It should also be noted that TEE_CopyOperation will also need the ability to return

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
					TEE_ERROR_OUT_OF_MEMORY.
27	146	6.2.4	E(T)	Undefined behavior. Function specification does not specify how operation size should be calculated (uint32_t *operationSize as function parameter). Should operation size be determined by Attributes? Attributes + operation handle? Else?	Specify exactly what is the required behavior.
28	144-147	6.2.3– 6.2.4	T	Redundancy. TEE_GetOperationInfo is subset (almost) from TEE_GetOperationInfoMultiple function. There is no point of two functions that fulfill the same task. In addition to this useful variables are scattered between both functions. Operation state could be useful at TEE_GetOperationInfo.	As a naming convention TEE_GetOperationInfo is more descriptive name than TEE_GetOperationInfoMultiple. Therefore TEE_GetOperationInfoMultiple functionality should be merged into TEE_GetOperationInfo. TEE_OperationInfo struct should be supplement with missing fields that existin the current TEE_OperationMultiple struct.
29	148	6.2.5	E	Helpful specification. If operation is only meaningful in a multistage operation, it should list operations. It would make it more clear for the TEE implementation and of course for user also, when there are exact definitions.	List operation types DIGEST, CIPHER and MAC → other types will be ignored and cause a panic.
30	148	6.2.5	T	Unnecessary panic reason. If key is not set for operation, response is panic. It seems like an excessive response? It would be more usable if TA would not panic. It does not reveal sensitive information about operation.	Remove panic, if key is not set and function returns to doing nothing.
31	149	6.2.6	T	Unexpected functionality. It should not be possible to set a new key without clearing the operation key first. This will make the interface more dynamic, but it would expose uncontrolled behavior to the user. The user should keep track of the operation state. This additional functionality adds complexity to the function and benefit is outweighed but the potential undesired behavior.	A new key can not be set for and operation if a key is already set. This should cause panic reason.
32	149	6.2.6	T	Hidden functionality. The clearing of a function key should be separated out into its own function as it is a distinctly separate operation.	Introduce new function TEE_ClearOperationKeys, which clear functions key. Function will set operation handle to state immediately after operation handle allocation. If TEE_SetOperationKey function is called when key is set → panic reason

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
33	151	6.2.7	T	Unexpected functionality. Apply same conclusion as point #39 and #40	Apply same proposition as point #39 and 40.
34	153	6.3	T	Uniform definition. It would allow more flexibility in the TEE implementation if there would be an initializion function for digest initialization.	Introduce TEE_DigestInit function that will initialize a digest operation.
35	154	6.3.2	T	Clarification. Apply same conclusion as point #38.	Apply same proposition as point #38.
36	155	6.4.1	T	Unexpected functionality. It should not be possible to initialize an operation if it is in active or initialized states. This will make the interface more dynamic, but it could raise uncontrolled behavior for the user. User should keep track of the operation state. Additional functionality adds complexity to function and the gained benefits are questionable.	If operation is active or initialized → panic. User should use TEE_ResetOperation function.
37	156	6.4.2	T	Clarification or unnecessary functionality. What is the meaning of no output is generated unless sufficient input is not supplied? For example should no pad algorithm source data be buffered if it is not block sized? If that is the case, this is unnecessary functionality. It is user responsibility to supply a correct size source buffer. At the end, user still must provide correct source buffer, before operation can be finalized. This adds complexity, which might be left “unsaid”.	Source buffer should be correct size. If not, it is a panic reason. Cryptographic failure.
38	157	6.4.3	T	Clarification. Function definition should instruct exactly what should happen in operation handler if operation is completed successfully. Loose definition leaves too much room for interpretation.	A good practice would be to instruct call TEE_ResetOperation function. Change to specification: “The operation handle can be reused or re-initialized” should be replaced with “If operation return code is TEE_SUCCESS, TEE_ResetOperation function is called.” This would make the statement “and is set to initial state afterward” obsolete and it could be removed.
39	158	6.5.1	T	Unexpected functionality. Apply same conclusion as point #36.	Apply same proposition as point #36.
40	158	6.5.2	T	Clarification or unnecessary functionality. Apply same conclusion as point #37.	Apply same proposition as point #37.

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
41	162	6.6	T	Security flaw. Authenticated encryption function is leaking plain text at decryption phase. A user may call TEE_AEUpdate multiple times in TEE_MODE_DECRYPT and this might pass back plain text prior to the MAC being verified. The flaw is pointed out in paper Authenticated Encryption Primitives for Size-Constrained Trusted Computing by Jan-Erik Ekberg, Alexandra Afanasyeva, and N. Asokan (http://dx.doi.org/10.1007/978-3-642-30921-2_1)	*Proposal 1*: Remove TEE_AEUpdate function and enforce use only secure TEE_AEDecryptFinal function. Function is secure, because you can only decrypt one piece cipher text (function accepting src data) and MAC can be checked prior to it's passing back. Notice: This force also at encryption phase use only TEE_AEEncrpytFinal. *Proposal 2*: Decryption is done in two stages (use decrypt-encrypt-decrypt strategy). TEE_AEUpdate function is decrypting source data, but does not pass plain text back to caller. It will encrypt plain text. When user calls TEE_DecryptFinal function, MAC will be verified. If MAC is okay, TEE_DecryptFinal function will reveal temporary key as one of the out parameters. With this key user can decrypt data to plain. Proposal 2 does not need big changes for API. It would be TEE implementation specification, how temporary key is generated and what algorithm is used to encrypt-decrypt, because this would be completely opaque to user. Changes to API: - Add return code TEE_AEDecryptFinal TEE_ERROR_OUT_OF_MEMORY, - Add out parameter to TEE_AEDecryptFinal function TEE_OperationHandle *decryptOperation. Operation handle is allocated and initialized according to encrypt operation. With this operation handler, user is able to decrypt data that he received at TEE_AEUpdate function. - Add explanation 6.6 chapter about decrypt-encrypt-decrypt strategy. Note: This is only a grandiose example. If you are interested about my idea, I can draft a more

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
					detailed version. My draft would follow this specification style.
42	162	6.6.1	T	Error. Subsequent calls are not possible, if operation state stays initial after TEE_AEInit function.	Afterward operation state should move to active state.
43	173	6.8.1	T	Undefined behavior. Object type is defined, but attribute type is not. If derived secret is used with symmetric operations, it can be guessed to TEE_ATTR_SECRET_VALUE. By placing the calculated value in TEE_ATTR_SECRET_VALUE it restricts the solution because the attribute is not extractable. This is constraining if you do not use the value in cryptographic operation.	Replace returned object handler with void pointer and 32-bit unsigned integer which is representing buffer length. Buffer is allocated by function and therefore function will be needing return code TEE_ERROR_OUT_OF_MEMORY.
44	173	6.8.1	T	Unnecessary complexity. Returning derived value in object handler is clumsy. See #43	See #43
45	188	7.2.4	T	Obsolete return codes. TA can have only one persistent time, it is feasible to suggest that the time variable is initialized when TA is loaded.	Remove all return codes and change function return value to void.
46	190	8	T	General. Big integer operation might require inner state. For example big integer negation. Having an opaque handler for Big integers allows for more flexibility in the TEE implementation. The granularity gained by allowing the user to managing memory directly is small, because it still requires the use of TEE_BigInt with defined function.	TEE_BigInt should handled in opaque way. API should define: typedef struct __TEE_BigInt *TEE_BigInt.
47	190	8	T	New subsection and functionality. TEE_BigInt will be needing allocation, initialization and free functions, if point 46 is agreed.	Add new subsection 8.5 Generic functions. - TEE_Result TEE_AllocateBigInt(TEE_BigInit *BigInt, uint32_t size): Allocated resources for TEE_BigInt and initializes meta structures. - Void TEE_BigInitInit(TEE_BigInt, bigInt, Void *bigInit, uint32_t length): Populate big init. -Void TEE_FreeBigInit(TEE_BigInit bigInt): Free resources -Return codes and panics: Same style as for example object handler or operation handler.
48	190	8	T	General. The inability for almost all of these functions to return a valid error is far too restrictive. There are numerous cases where an error could occur and this information should	Functions, which return type is void, should be replaced with TEE_Result. In case of success the return code is TEE_SUCCESS and in case

#	Page#	Chapter / Paragraph (*)	Type(**)	Comment (Problem & Reason)	Proposed Resolution
				be conveyed to the caller of the function. An operation can fail due to a number of reasons including, out of memory, if temporary buffer is needed.	of any error the return code is TEE_ERROR_GENERIC or a set of error codes defined that match the possible fault conditions.
49	201	8.7	T	Useful function. No bit set function.	Introduce functionality for setting a bit in a Big Integer: - Void TEE_BigIntSetBit(TEE_BigInt *op, uint32_t bitPos, uint32_t setBit);
50	-	-	T	New concept. If object allocation could be made more dynamic it would be possible to review the *maxKeySize*-concept. From security point of view it is arguable if you handle max key strength rather than min key strength. In min strength enforcing you always use “secure” key.	Change key *maxKeysize* to *minKeySize* and definitions at functions where value is used.
51	-	-	T	Good practice. TEE_FreeTransientObject, TEE_ObjectClose, TEE_CloseAndDeletePersistentObject, TEE_FreePersistentObjectEnumerator and TEE_FreeOperation should always set operation/object/enumeration to NULL.	Add “The value pointed to by object/operation/enumeration is set to TEE_HANDLE_NULL” to the function specification.
52	-	6.1	E	Reordering document. If new subsection (6.2 Constant) is defined, it could also contain chapter 5.4 operation related constants. By this operation constant finds its own chapter -> more readable.	Move table 5-6 to new subsection 6.2

(*)(e.g. 5.3 §2 – line 6)
(**)T=Technical, E=Editorial

Each member is bound by the terms of the current GlobalPlatform IPR Policy, a copy of which is available on both the Public and Member websites and is available upon request from the Secretariat. In addition, all non-members submitting Comments acknowledge and agree to ad999here to the current GlobalPlatform IPR Policy.